10

Building Multimodal Applications with LLMs

In this chapter, we are going beyond LLMs, to introduce the concept of multimodality while building agents. We will see the logic behind the combination of foundation models in different AI domains – language, images, and audio – into one single agent that can adapt to a variety of tasks. By the end of this chapter, you will be able to build your own multimodal agent, providing it with the tools and LLMs needed to perform various AI tasks.

Throughout this chapter, we will cover the following topics:

- Introduction to multimodality and **large multimodal models** (**LMMs**)
- Examples of emerging LMMs
- How to build a multimodal agent with single-modal LLMs using LangChain

Technical requirements

To complete the tasks in this chapter, you will need the following:

- A Hugging Face account and user access token.
- An OpenAI account and user access token.
- Python 3.7.1 or later version.
- Python packages. Make sure to have the following Python packages installed: `langchain`, `python-dotenv`, `huggingface_hub`, `streamlit`, `pytube`, `openai`, and `youtube_search`. Those can be easily installed via `pip install` in your terminal.

You can find all the code and examples in the book's GitHub repository at https://github.com/PacktPublishing/Building-LLM-Powered-Applications.

Why multimodality?

In the context of Generative AI, multimodality refers to a model's capability of processing data in various formats. For example, a multimodal model can communicate with humans via text, speech, images, or even videos, making the interaction extremely smooth and "human-like."

In *Chapter 1*, we defined **large foundation models** (**LFMs**) as a type of pre-trained generative AI model that offers immense versatility by being adaptable for various specific tasks. LLMs, on the other hand, are a subset of foundation models that are able to process one type of data: natural language. Even though LLMs have proven to be not only excellent text understanders and generators but also reasoning engines to power applications and copilots, it soon became clear that we could aim at even more powerful applications.

The dream is to have intelligent systems that are capable of handling multiple data formats – text, images, audio, video, etc – always powered by the reasoning engine, which makes them able to plan and execute actions with an agentic approach. Such an AI system would be a further milestone toward the reaching of **artificial general intelligence** (**AGI**).

Definition

AGI is a hypothetical type of **artificial intelligence** (**AI**) that can perform any intellectual task that a human can. AGI would have a general cognitive ability, similar to human intelligence, and be able to learn from experience, reason, plan, communicate, and solve problems across different domains. An AGI system would also be able to "perceive" the world as we do, meaning that it could process data in different formats, from text to images to sounds. Hence, AGI implies multimodality.

Creating AGI is a primary goal of some AI research and a common topic in science fiction. However, there is no consensus on how to achieve AGI, what criteria to use to measure it, or when it might be possible. Some researchers argue that AGI could be achieved in years or decades, while others maintain that it might take a century or longer, or that it might never be achieved.

However, AGI is not seen as the ultimate milestone in AI development. In fact, in recent months another definition has emerged in the context of AI – that is, Strong AI or Super AI, referring to an AI system that is more capable than a human.

At the time of writing this book (February 2024), LMMs such as GPT-4 Turbo with Vision are a reality. However, those are not the only ways to reach multimodality. In this chapter, we are going to examine how to merge multiple AI systems to reach a multimodal AI assistant. The idea is that if we combine single-modal models, one for each data format we want to process, and then use an LLM as the brain of our agent to let it interact in dynamic ways with those models (that will be its tools), we can still achieve this goal. The following diagram shows the structure of a multimodal application that integrates various single-modal tools to perform a task – in this case, describing a picture aloud. The application uses image analysis to examine the picture, text generation to create some text that describes what it observes in the picture, and text-to-speech to convey this text to the user through speech.

The LLM acts as the "reasoning engine" of the application, invoking the proper tools needed to accomplish the user's query.

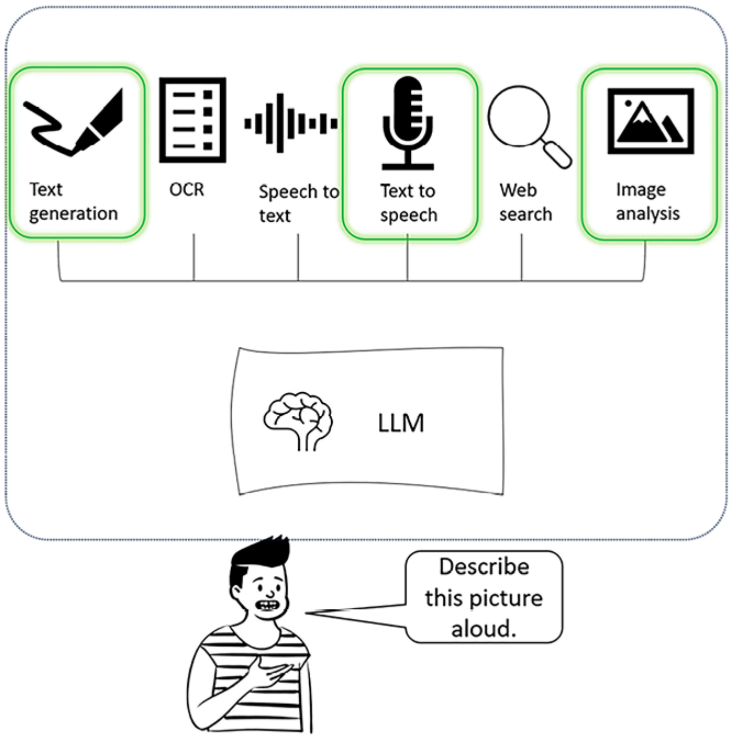


Figure 10.1: Illustration of multimodal application with single-modal tools

In the upcoming section, we are going to explore various approaches to building multimodal applications, all based on the idea of combining existing single-modal tools or models.

Building a multimodal agent with LangChain

So far, we've covered the main aspects of multimodality and how to achieve it with modern LFMs. As we saw throughout Part 2 of this book, LangChain offers a variety of components that we leveraged massively, such as chains, agents, tools, and so on. As a result, we already have all the ingredients we need to start building our multimodal agent.

However, in this chapter, we will adopt three approaches to tackle the problem:

- **The agentic, out-of-the-box approach**: Here we will leverage the Azure Cognitive Services toolkit, which offers native integrations toward a set of AI models that can be consumed via API, and that covers various domains such as image, audio, OCR, etc.
- **The agentic, custom approach**: Here, we are going to select single models and tools (including defining custom tools) and concatenate them into a single agent that can leverage all of them.
- **The hard-coded approach**: Here, we are going to build separate chains and combine them into a sequential chain.

In the upcoming sections, we will cover all these approaches with concrete examples.

Option 1: Using an out-of-the-box toolkit for Azure AI Services

Formerly known as Azure Cognitive Services, Azure AI Services are a set of cloud-based APIs and AI services developed by Microsoft that enable developers and data scientists to add cognitive capabilities to their apps. AI Services are meant to provide every developer with AI models to be integrated with programming languages such as Python, C#, or JavaScript.

Azure AI Services cover various domains of AI, including speech, natural language, vision, and decision-making. All those services come with models that can be consumed via API, and you can decide to:

- Leverage powerful pre-built models available as they are and ready to use.
- Customize those pre-built models with custom data so that they are tailored to your use case.

Hence, considered all together, Azure AI Services can achieve the goal of multimodality, if properly orchestrated by an LLM as a reasoning engine, which is exactly the framework LangChain built.

Getting Started with AzureCognitiveServicesToolkit

In fact, LangChain has a native integration with Azure AI Services called **AzureCognitiveServicesToolkit**, which can be passed as a parameter to an agent and leverage the multimodal capabilities of those models.

The toolkit makes it easier to incorporate Azure AI services' capabilities – such as image analysis, form recognition, speech-to-text, and text-to-speech – within your application. It can be used within an agent, which is then empowered to use the AI services to enhance its functionality and provide richer responses.

Currently, the integration supports the following tools:

- **AzureCogsImageAnalysisTool**: Used to analyze and extract metadata from images.
- **AzureCogsSpeech2TextTool**: Used to convert speech to text.
- **AzureCogsText2SpeechTool**: Used to synthetize text to speech with neural voices.
- **AzureCogsFormRecognizerTool**: Used to perform **optical character recognition** (**OCR**).

Definition

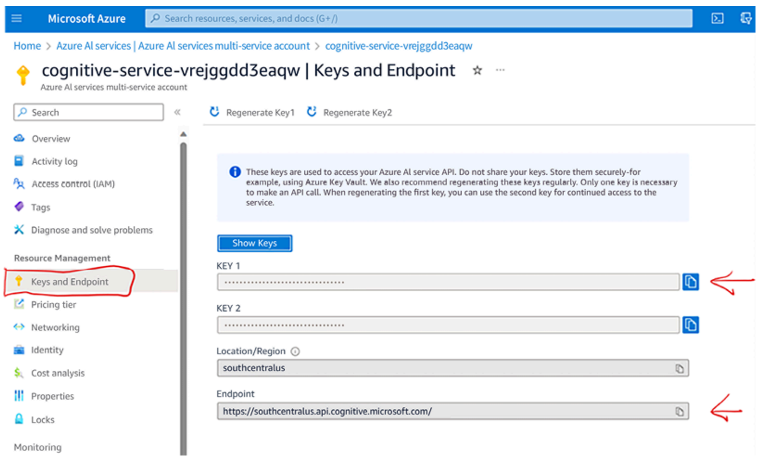
OCR is a technology that converts different types of documents, such as scanned paper documents, PDFs, or images captured by a digital camera, into editable and searchable data. OCR can save time, cost, and resources by automating data entry and storage processes. It can also enable access to and editing of the original content of historical, legal, or other types of documents.

For example, if you ask an agent what you can make with some ingredients, and provide an image of eggs and flour, the agent can use the Azure AI Services Image Analysis tool to extract the caption, objects, and tags from the image, and then use the provided LLM to suggest some recipes based on the ingredients. To implement this, let's first set up our toolkit.

Setting up the toolkit

To get started with the toolkit, you can follow these steps:

1. You first need to create a multi-service instance of Azure AI Services in Azure following the instructions at https://learn.microsoft.com/en-us/azure/ai-services/multi-service-resource?tabs=windows&pivots=azportal.
2. A multi-service resource allows you to access multiple AI services with a single key and endpoint to be passed to LangChain as environmental variables. You can find your keys and endpoint under the **Keys and Endpoint** tab in your resource panel:



cognitive-service-vrejggdd3eaqw | Keys and Endpoint

Azure AI services multi-service account

These keys are used to access your Azure AI service API. Do not share your keys. Store them securely-for example, using Azure Key Vault. We also recommend regenerating these keys regularly. Only one key is necessary to make an API call. When regenerating the first key, you can use the second key for continued access to the service.

KEY 1: ································

KEY 2: ································

Location/Region: southcentralus

Endpoint: https://southcentralus.api.cognitive.microsoft.com/

Figure 10.2: Screenshot of a multi-service instance of Azure AI Services

3. Once the resource is set, we can start building our LegalAgent. To do so, the first thing we need to do is set the AI services environmental variables in order to configure the toolkit. To do so, I've saved the following variables in my `.env` file:

```
AZURE_COGS_KEY = "your-api-key"
AZURE_COGS_ENDPOINT = "your-endpoint
AZURE_COGS_REGION = "your-region"
```

4. Then, you can load them as always alongside the other environmental variables:

```python
import os
from dotenv import load_dotenv
load_dotenv()
azure_cogs_key = os.environ["AZURE_COGS_KEY"]
azure_cogs_endpoint = os.environ["AZURE_COGS_ENDPOINT"]
azure_cogs_region = os.environ["AZURE_COGS_REGION"]
openai_api_key = os.environ['OPENAI_API_KEY']
```

5. Now, we can configure our toolkit and also see which tools we have, alongside their description:

```
from langchain.agents.agent_toolkits import AzureCognitiveServicesToolkit
toolkit = AzureCognitiveServicesToolkit()
[(tool.name, tool.description) for tool in toolkit.get_tools()]
```

The following is the corresponding output:

```
[('azure_cognitive_services_form_recognizer',
  'A wrapper around Azure Cognitive Services Form Recognizer. Useful for when you need to
 ('azure_cognitive_services_speech2text',
  'A wrapper around Azure Cognitive Services Speech2Text. Useful for when you need to tran
 ('azure_cognitive_services_text2speech',
  'A wrapper around Azure Cognitive Services Text2Speech. Useful for when you need to conv
 ('azure_cognitive_services_image_analysis',
  'A wrapper around Azure Cognitive Services Image Analysis. Useful for when you need to a
```

6. Now, it's time to initialize our agent. For this purpose, we will use a `STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION` agent that, as we saw in previous chapters, also allows for multi-tools input, since we will also add further tools in the *Leveraging multiple tools* section:

```
from langchain.agents import initialize_agent, AgentType
from langchain import OpenAI
llm = OpenAI()
Model = ChatOpenAI()
agent = initialize_agent(
    tools=toolkit.get_tools(),
    llm=llm,
    agent=AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True,
)
```

Now we have all the ingredients to start testing our agent.

Leveraging a single tool

To start easy, let's simply ask the agent to describe the following picture, which will only require the `image_analysis` tool to be accomplished:

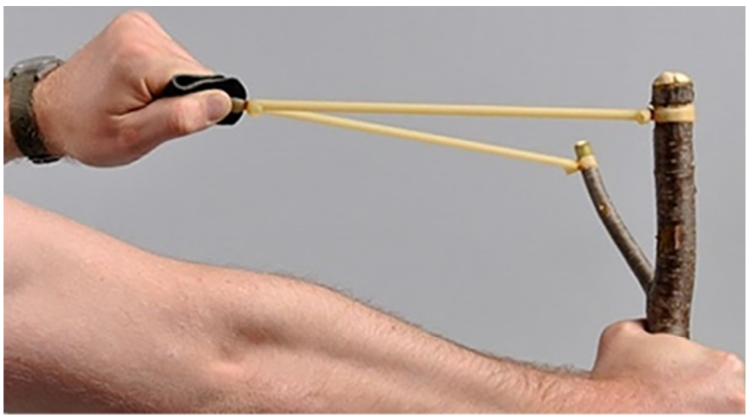


Figure 10.3: Sample picture of a slingshot (source: https://www.stylo24.it/wp-content/uploads/2020/03/fionda.jpg*)*

Let's pass the URL of this image as input to our model, as per the description of the `azure_cognitive_services_image_analysis` tool:

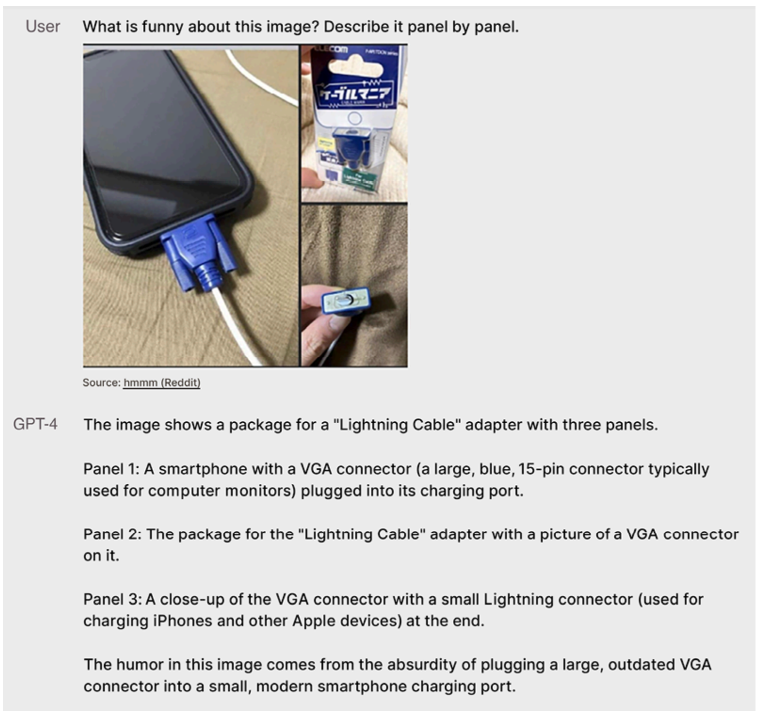
```
description = agent.run("what shows the following image?:"
    "https://www.stylo24.it/wp-content/uploads/2020/03/fionda.jpg")
print(description)
```

We then get the following output:

```
> Entering new AgentExecutor chain...
Action:
```
{
  "action": "azure_cognitive_services_image_analysis",
  "action_input": "https://www.stylo24.it/wp-content/uploads/2020/03/fionda.jpg"
}
```
Observation: Caption: a person holding a slingshot
Tags: person, tool, nail, hand, holding, needle
Thought: I know what the image is.
Action:
```
{
  "action": "Final Answer",
  "action_input": "The image is of a person holding a slingshot."
}
```
> Finished chain.
The image is of a person holding a slingshot.
```

As you can see, the agent was able to retrieve the proper tool to address the user's question. In this case, the question was very simple, so I want to challenge the same tool with a trickier question.

The goal is to replicate the GPT-4 capabilities in its common-sense reasoning while working with images, as the following illustration from GPT-4's earliest experiments shows:



User What is funny about this image? Describe it panel by panel.

Source: hmmm (Reddit)

GPT-4 The image shows a package for a "Lightning Cable" adapter with three panels.

Panel 1: A smartphone with a VGA connector (a large, blue, 15-pin connector typically used for computer monitors) plugged into its charging port.

Panel 2: The package for the "Lightning Cable" adapter with a picture of a VGA connector on it.

Panel 3: A close-up of the VGA connector with a small Lightning connector (used for charging iPhones and other Apple devices) at the end.

The humor in this image comes from the absurdity of plugging a large, outdated VGA connector into a small, modern smartphone charging port.

*Figure 10.4: Example of visual capabilities and common sense reasoning of GPT-4 (source: **https://openai.com/research/gpt-4**)*

So let's ask our model something more challenging. Let's ask it to reason about the consequences of letting the slingshot go:

```
agent.run("what happens if the person lets the slingshot go?:"
    "https://www.stylo24.it/wp-content/uploads/2020/03/fionda.jpg")
```

We then obtain the following output:

```
> Entering new AgentExecutor chain...
Action:
```
{
  "action": "azure_cognitive_services_image_analysis",
  "action_input": "https://www.stylo24.it/wp-content/uploads/2020/03/fionda.jpg"
}
```
Observation: Caption: a person holding a slingshot
Tags: person, tool, nail, hand, holding, needle
Thought: I know what to respond
Action:
```
{
  "action": "Final Answer",
  "action_input": "If the person lets the slingshot go, it will fly through the air."
}
```
> Finished chain.
'If the person lets go of the slingshot, the object being launched by it would be released
```

It might seem an easy question, but the agent's answers imply an extremely refined common sense reasoning: thanks to the metadata extracted from the image leveraging the `image_analysis` tool, the LLM was able to set up reasoning in terms of consequences given an action (the person letting the slingshot go).

In this example, the agent only leveraged one tool. But what happens if we ask it something that requires at least two tools?

Leveraging multiple tools

Let's say, for example, that we want the model to read a story aloud to us based on a picture.

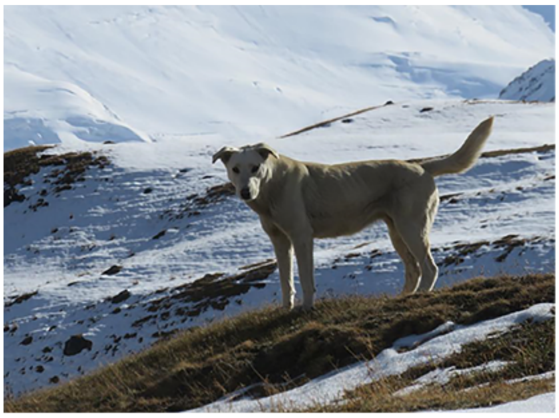


Figure 10.5: Example of an input image for a story-telling agent (source: https://i.redd.it/diawvlriobq11.jpg)

Let's ask the agent to do so:

```
agent.run("Tell me a story related to the following picture and read the story aloud to me
```

We obtain the following output:

```
Thought:The human requested a story related to the picture. I will use the Azure Cognitive
Action:
```
{
  "action": "azure_cognitive_services_image_analysis",
  "action_input": {
    "query": "https://i.redd.it/diawvlriobq11.jpg"
  }
}
```
Observation: Caption: a dog standing on a snowy hill
Objects: dog
Tags: outdoor, animal, mammal, snow, freezing, dog breed, dog, mountain, covered, winter
Thought:Based on the Azure Cognitive Services Image Analysis tool's analysis of the pictur
Action:
```
{
  "action": "azure_cognitive_services_text2speech",
  "action_input": {
    "query": "Once upon a time, in a snowy winter wonderland, there lived a brave little d
  }
}
```
Observation: C:\Users\vaalt\AppData\Local\Temp\tmp_cecwqht.wav
Thought:The story involving a dog in a snowy winter wonderland seems to have satisfied the
Action:
```
{
  "action": "Final Answer",
  "action_input": "I hope you enjoyed the story of Snowy the Savior of the Snow, the brave
}
```
```

As you can see, the agent was able to invoke two tools to accomplish the request:

1. It first started with the `image_analysis` tool to generate the image caption used to produce the story.
2. Then, it invoked the `text2speech` tool to read it aloud to the user.

The agent saved the audio file in a temporary file, and you can listen to it directly by clicking on the URL. Alternatively, you can save the output as a Python variable and execute it as follows:

```
from IPython import display
audio = agent.run("Tell me a story related to the following picture and read the story alo
display.display(audio)
```

Finally, we can also modify the default prompt that comes with the agent type, to make it more customized with respect to our specific use case. To do so, we first need to inspect the template and then decide which part we can modify. To inspect the template, you can run the following command:

```
print(agent.agent.llm_chain.prompt.messages[0].prompt.template)
```

Here is our output:

```
Respond to the human as helpfully and accurately as possible. You have access to the follo
{tools}
Use a json blob to specify a tool by providing an action key (tool name) and an action_inp
Valid "action" values: "Final Answer" or youtube_search, CustomeYTTranscribe
Provide only ONE action per $JSON_BLOB, as shown:
```
{{
  "action": $TOOL_NAME,
  "action_input": $INPUT
}}
```
Follow this format:
Question: input question to answer
Thought: consider previous and subsequent steps
Action:
```
$JSON_BLOB
...
```
Begin! Reminder to ALWAYS respond with a valid json blob of a single action. Use tools if
Thought:
```

Let's modify the prefix of the prompt and pass it as `kwargs` to our agent:

```
PREFIX = """
You are a story teller for children.
You read aloud stories based on pictures that the user pass you.
 You always start your story with a welcome message targeting children, with the goal of m
 You can use multiple tools to answer the question.
 ALWAYS use the tools.
 You have access to the following tools:"""
agent = initialize_agent(toolkit.get_tools(), model, agent=AgentType.STRUCTURED_CHAT_ZERO_
                         agent_kwargs={
                            'prefix':PREFIX})
```

As you can see, now the agent acts more similar to a storyteller with a specific style. You can customize your prompt as you wish, always keeping in mind that each pre-built agent has its own prompt template, hence it is always recommended to first inspect it before customizing it.

Now that we have explored the out-of-the-box capabilities of the toolkit, let's build an end-to-end application.

Building an end-to-end application for invoice analysis

Analyzing invoices might require a lot of manual work if not assisted by digital processes. To address this, we will build an AI assistant that is able to analyze invoices for us and tell us any relevant information aloud. We will call this application **CoPenny**.

With CoPenny, individuals and enterprises could reduce the time of invoice analysis, as well as build toward document process automation and, more generally, digital process automation.

Definition

Document process automation is a strategy that uses technology to streamline and automate various document-related tasks and processes within an organization. It involves the use of software tools, including document capture, data extraction, workflow automation, and integration with other systems. For example, document process automation can help you extract, validate, and analyze data from invoices, receipts, forms, and other types of documents. Document process automation can save you time and money, improve accuracy and efficiency, and provide valuable insights and reports from your document data.

Digital process automation (**DPA**) is a broader term that refers to automating any business process with digital technology. DPA can help you connect your apps, data, and services and boost your team's productivity with cloud flows. DPA can also help you create more sophisticated and intuitive customer experiences, collaborate across your organization, and innovate with AI and ML.

To start building our application, we can follow these steps:

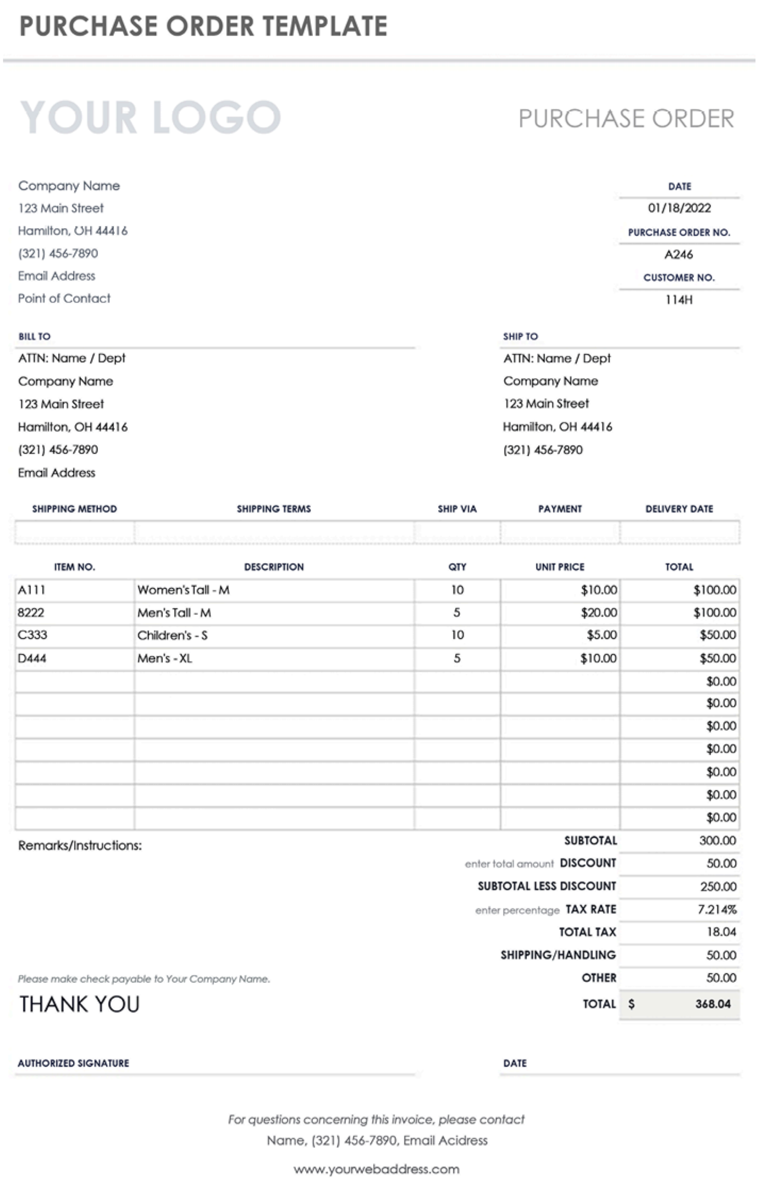
1. Using `AzureCognitiveServicesToolkit`, we will leverage the `azure_cognitive_services_form_recognizer` and `azure_cognitive_services_text2speech` tools, so we can limit the agent's "powers" only to those two:

```
toolkit = AzureCognitiveServicesToolkit().get_tools()
#those tools are at the first and third position in the list
tools = [toolkit[0], toolkit[2]]
tools
```

The following is the corresponding output:

```
[AzureCogsFormRecognizerTool(name='azure_cognitive_services_form_recognizer', description=
```

2. Let's now initialize the agent with the default prompt and see the results. For this purpose, we will use a sample invoice as a template with which to query the agent:


PURCHASE ORDER TEMPLATE

YOUR LOGO

PURCHASE ORDER

Company Name
123 Main Street
Hamilton, OH 44416
(321) 456-7890
Email Address
Point of Contact

DATE	01/18/2022
PURCHASE ORDER NO.	A246
CUSTOMER NO.	114H

BILL TO	SHIP TO
ATTN: Name / Dept	ATTN: Name / Dept
Company Name	Company Name
123 Main Street	123 Main Street
Hamilton, OH 44416	Hamilton, OH 44416
(321) 456-7890	(321) 456-7890
Email Address	

SHIPPING METHOD	SHIPPING TERMS	SHIP VIA	PAYMENT	DELIVERY DATE

ITEM NO.	DESCRIPTION	QTY	UNIT PRICE	TOTAL
A111	Women's Tall - M	10	$10.00	$100.00
8222	Men's Tall - M	5	$20.00	$100.00
C333	Children's - S	10	$5.00	$50.00
D444	Men's - XL	5	$10.00	$50.00
				$0.00
				$0.00
				$0.00
				$0.00
				$0.00
				$0.00
				$0.00

Remarks/Instructions:

SUBTOTAL	300.00
enter total amount DISCOUNT	50.00
SUBTOTAL LESS DISCOUNT	250.00
enter percentage TAX RATE	7.214%
TOTAL TAX	18.04
SHIPPING/HANDLING	50.00
OTHER	50.00
TOTAL	$ 368.04

Please make check payable to Your Company Name.

THANK YOU

AUTHORIZED SIGNATURE

DATE

For questions concerning this invoice, please contact
Name, (321) 456-7890, Email Address
www.yourwebaddress.com

Figure 10.6: Sample template of a generic invoice (source: https://www.whiteelysee.fr/design/wp-content/uploads/2022/01/custom-t-shirt-order-form-template-free.jpg)

3. Let's start by asking the model to tell us all the men's **stock-keeping units** (**SKUs**) on the invoice:

```
agent.run("what are all men's skus?"
    "https://www.whiteelysee.fr/design/wp-content/uploads/2022/01/custom-t-shirt-order-
```

We then get the following output (showing a truncated output; you can find the whole output in the book's GitHub repository):

~~~
> Entering new AgentExecutor chain...
Action:
```
{
  "action": "azure_cognitive_services_form_recognizer",
  "action_input": {
    "query": "https://www.whiteelysee.fr/design/wp-content/uploads/2022/01/custom-t-shirt-
  }
}
```
~~~

```
Observation: Content: PURCHASE ORDER TEMPLATE […]
> Finished chain.
"The men's skus are B222 and D444."
```

4. We can also ask for multiple information (women's SKUs, shipping address, and delivery dates) as follows (note that the delivery date is not specified, as we want our agent not to hallucinate):

```
agent.run("give me the following information about the invoice: women's SKUs, shipping
    "https://www.whiteelysee.fr/design/wp-content/uploads/2022/01/custom-t-shirt-order-
```

This gives us the following output:

```
"The women's SKUs are A111 Women's Tall - M. The shipping address is Company Name 123 Main
```

5. Finally, let's also leverage the text2speech tool to produce the audio of the response:

```
agent.run("extract women's SKUs in the following invoice, then read it aloud:"
    "https://www.whiteelysee.fr/design/wp-content/uploads/2022/01/custom-t-shirt-order-
```

As per the previous example, you can listen to the audio by clicking on the URL in the chain, or using Python's `Display` function if you save it as a variable.

6. Now, we want our agent to be better tailored toward our goal. To do so, let's customize the prompt giving specific instructions. In particular, we want the agent to produce the audio output without the user explicitly asking for it:

```
PREFIX = """
You are an AI assistant that help users to interact with invoices.
You extract information from invoices and read it aloud to users.
You can use multiple tools to answer the question.
Always divide your response in 2 steps:
1. Extracting the information from the invoice upon user's request
2. Converting the transcript of the previous point into an audio file
ALWAYS use the tools.
ALWAYS return an audio file using the proper tool.
You have access to the following tools:
"""
agent = initialize_agent(tools, model, agent=AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_
                         agent_kwargs={
                            'prefix':PREFIX})
```

7. Let's run the agent:

```
agent.run("what are women's SKUs in the following invoice?:"
    "https://www.whiteelysee.fr/design/wp-content/uploads/2022/01/custom-t-shirt-order-
```

This yields the following output:

```
> Entering new AgentExecutor chain...
I will need to use the azure_cognitive_services_form_recognizer tool to extract the inform
```

```
Action:
```
{
  "action": "azure_cognitive_services_form_recognizer",
  "action_input": {
    "query": "https://www.whiteelysee.fr/design/wp-content/uploads/2022/01/custom-t-shirt-
  }
}
```
Observation: Content: PURCHASE ORDER TEMPLATE […]
Observation: C:\Users\vaalt\AppData\Local\Temp\tmpx1n4obf3.wav
Thought:Now that I have provided the answer, I will wait for further inquiries.
```

As you can see, now the agent saved the output into an audio file, even when the user didn't ask explicitly for it.

`AzureCognitiveServicesToolkit` is a powerful integration that allows for native consumption of Azure AI Services. However, there are some pitfalls of this approach, including the limited number of AI services. In the next section, we are going to explore yet another option to achieve multimodality, with a more flexible approach while still keeping an agentic strategy.

Option 2: Combining single tools into one agent

In this leg of our journey toward multimodality, we will leverage different tools as plug-ins to our `STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION` agent. Our goal is to build a copilot agent that will help us generate reviews about YouTube videos, as well as post those reviews on our social media with a nice description and related picture. In all of that, we want to make little or no effort, so we need our agent to perform the following steps:

1. Search and transcribe a YouTube video based on our input.
2. Based on the transcription, generate a review with a length and style defined by the user query.
3. Generate an image related to the video and the review.

We will call our copilot **GPTuber**. In the following subsections, we will examine each tool and then put them all together.

YouTube tools and Whisper

The first step of our agent will be to search and transcribe the YouTube video based on our input. To do so, there are two tools we need to leverage:

- **YouTubeSearchTool**: An out-of-the-box tool offered by LangChain and adapted from **https://github.com/venuv/langchain_yt_tools**. You can import and try the tool by running the following code, specifying the topic of the video and the number of videos you want the tool to return:

  ```
  from langchain.tools import YouTubeSearchTool
  tool = YouTubeSearchTool()
  ```

```
result = tool.run("Avatar: The Way of Water,1")
result:
```

Here is the output:

```
"['/watch?v=d9MyW72ELq0&pp=ygUYQXZhdGFyOiBUaGUgV2F5IG9mIFdhdGVy']"
```

The tool returns the URL of the video. To watch it, you can add it to **https://youtube.com domain**.

- **CustomYTTranscribeTool**: This is a custom tool that I've adapted from **https://github.com/venuv/langchain_yt_tools**. It consists of transcribing the audio file retrieved from the previous tool using a speech-to-text model. In our case, we will be leveraging OpenAI's **Whisper**.

Whisper is a transformer-based model introduced by OpenAI in September 2022. It works as follows:

1. It splits the input audio into 30-second chunks, converting them into spectrograms (visual representations of sound frequencies).
2. It then passes them to an encoder.
3. The encoder then produces a sequence of hidden states that capture the information in the audio.
4. A decoder then predicts the corresponding text caption, using special tokens to indicate the task (such as language identification, speech transcription, or speech translation) and the output language.
5. The decoder can also generate timestamps for each word or phrase in the caption.

Unlike most OpenAI models, Whisper is open-source.

Since this model takes as input only files and not URLs, within the custom tool, there is a function defined as `yt_get` (you can find it in the GitHub repository) that, starting from the video URL, downloads it into a `.mp4` file. Once downloaded, you can try Whisper with the following lines of code:

```
import openai
audio_file = open("Avatar The Way of Water  Official Trailer.mp4", 'rb')
result = openai.Audio.transcribe("whisper-1", audio_file)
audio_file.close()
print(result.text)
```

Here is the corresponding output:

```
♪ Dad, I know you think I'm crazy. But I feel her. I hear her heartbeat. She's so close. ♪
```

By embedding Whisper in this custom tool, we can transcribe the output of the first tool into a transcript that will serve as input to the next tool. You can see the code and logic behind this embedding and the whole tool in this book's GitHub repository at **https://github.com/PacktPublishing/Building-LLM-Powered-Applications**, which is a modified version from **https://github.com/venuv/langchain_yt_tools**.

Since we already have two tools, we can start building our tools list and initializing our agent, using the following code:

```
llm = OpenAI(temperature=0)
tools = []
tools.append(YouTubeSearchTool())
tools.append(CustomYTTranscribeTool())
agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=
agent.run("search a video trailer of Avatar: the way of water. Return only 1 video. transc
```

The following is the corresponding output:

```
> Entering new AgentExecutor chain...
I need to find a specific video and transcribe it.
Action: youtube_search
Action Input: "Avatar: the way of water,1"
Observation: ['/watch?v=d9MyW72ELq0&pp=ygUYQXZhdGFyOiB0aGUgd2F5IG9mIHdhdGVy']
Thought:I found the video I was looking for, now I need to transcribe it.
Action: CustomeYTTranscribe
Action Input: […]
Observation: ♪ Dad, I know you think I'm crazy. […]
Thought:I have the transcription of the video trailer for Avatar: the way of water.
Final Answer: The transcription of the video trailer for Avatar: the way of water is: "♪ D
> Finished chain.
```

Great! We were able to generate the transcription of this video. The next step will be to generate a review alongside a picture. While the review can be written directly from the LLM and passed as a parameter to the model (so we don't need another tool), the image generation will need an additional tool. For this purpose, we are going to use OpenAI's DALL·E.

DALL·E and text generation

Introduced by OpenAI in January 2021, DALL·E is a transformer-based model that can create images from text descriptions. It is based on GPT-3, which is also used for natural language processing tasks. It is trained on a large dataset of text-image pairs from the web and uses a vocabulary of tokens for both text and image concepts. DALL·E can produce multiple images for the same text, showing different interpretations and variations.

LangChain offers native integration with DALL·E, which you can use as a tool by running the following code (always setting the environmental variable of your `OPENAI_API_KEY` from the `.env` file):

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
tools = load_tools(['dalle-image-generator'])
agent = initialize_agent(tools, model, AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True
agent.run("Create an image of a halloween night. Return only the image url.")
```

Here is the corresponding output:

```
> Entering new AgentExecutor chain...
I need to use an image generator to create an image of a halloween night.
Action: Dall-E Image Generator
Action Input: "An image of a spooky halloween night with a full moon, bats flying in the s
```

```
Observation: [link_to_the_blob]
Thought:I have successfully generated an image of a halloween night.
Final Answer: The image url is [link_to_the_blob]
> Finished chain.
```

The following is the image that was generated, as requested:

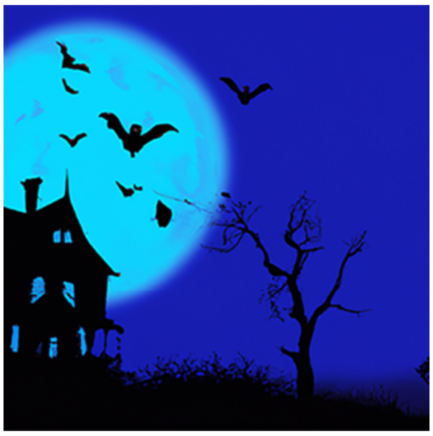


Figure 10.7: Image generated by DALL·E upon the user's input

Great! Now let's also see whether our agent is capable of generating a review of a video based on the transcription:

```
agent.run("search a video trailer of Avatar: the way of water. Return only 1 video. transc
```

We get the following output:

```
> Entering new AgentExecutor chain...
I need to use both youtube_search and CustomeYTTranscribe tools to answer this question.
Action: youtube_search
Action Input: "Avatar: the way of water,1"
Observation: ['/watch?v=d9MyW72ELq0&pp=ygUYQXZhdGFyOiB0aGUgd2F5IG9mIHdhdGVy']
Thought:I found the video I need to transcribe.
Action: CustomeYTTranscribe
Action Input: […]
Observation: ♪ Dad, I know you think I'm crazy. […]
Thought:I need to write a review of the trailer.
Action: Write a review
Action Input: "Avatar: The Way of Water trailer is visually stunning and captivating. The
Observation: Write a review is not a valid tool, try one of [youtube_search, CustomeYTTran
Thought:I need to use a different tool to write a review.
Action: Write a review
Action Input: N/A
Observation: Write a review is not a valid tool, try one of [youtube_search, CustomeYTTran
...
Thought:I can write the review manually.
Final Answer: Avatar: The Way of Water trailer is visually stunning and captivating. The t
> Finished chain.
```

Note how the agent was initially looking for a tool to make a review, to then realize that there is no tool yet that can do it manually thanks to its parametric knowledge. This is a great example of how LLMs are reasoning engines and endowed with common sense reasoning. As always, you can find the entire chain of thoughts in the book's repository.

The next step will be to put it all together and see whether the agent is capable of orchestrating all the tools, with some assistance in terms of prompt engineering.

Putting it all together

Now that we have all the ingredients, we need to put them together into one single agent. To do so, we can follow these steps:

1. First, we need to add the DALL·E tool to the list of tools:

   ```
   tools = []
   tools.append(YouTubeSearchTool())
   tools.append(CustomYTTranscribeTool())
   tools.append(load_tools(['dalle-image-generator'])[0])
   [tool.name for tool in tools]
   ```

This gives us the following output:

```
['youtube_search', 'CustomeYTTranscribe', 'Dall-E Image Generator']
```

2. The next step will be to test the agent with the default prompt, and then try to refine the instructions with some prompt engineering. Let's start with a pre-configured agent (you can find all the steps in the GitHub repository):

   ```
   agent = initialize_agent(tools, model, AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=T
   agent.run("search a video trailer of Avatar: the way of water. Return only 1 video. tra
   ```

This gives us the following output:

```
> Entering new AgentExecutor chain...
I need to search for a video trailer of "Avatar: The Way of Water" and transcribe it to ge
Action: youtube_search
Action Input: "Avatar: The Way of Water trailer,1"
Observation: ['/watch?v=d9MyW72ELq0&pp=ygUgQXZhdGFyOiBUaGUgV2F5IG9mIFdhdGVyIHRyYWlsZXI%3D'
Thought:I found a video trailer of "Avatar: The Way of Water" with the given search query.
Action: CustomeYTTranscribe
Action Input: '/watch?v=d9MyW72ELq0&pp=ygUgQXZhdGFyOiBUaGUgV2F5IG9mIFdhdGVyIHRyYWlsZXI%3D'
Observation: ♪ Dad, I know you think I'm crazy. […]
Thought:I have transcribed the video and now I can generate a review based on the transcri
Action: Dall-E Image Generator
Action Input: "Review of Avatar: The Way of Water trailer: The trailer for Avatar: The Way
Observation: [image_url]
Thought:I have generated a review of the "Avatar: The Way of Water" trailer based on the t
Final Answer: The "Avatar: The Way of Water" trailer is captivating and mysterious, featur
> Finished chain.
```

The following is the accompanying visual output:

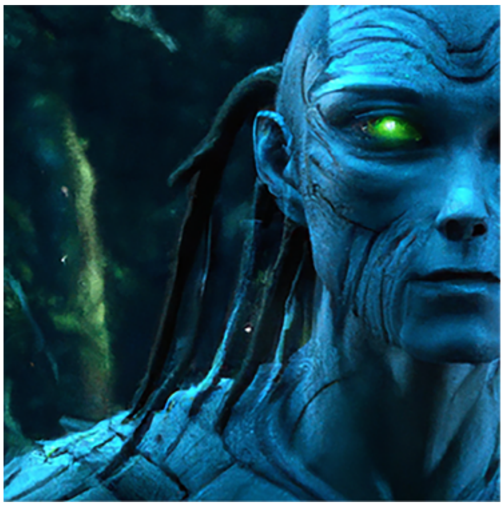


Figure 10.8: Image generated by DALL·E based on the trailer review

Well, even without any prompt engineering, the agent was able to orchestrate the tools and return the desired results!

3. Now, let's try to make it more tailored toward our purpose. Similar to the CoPenny application, we don't want the user to specify every time to generate a review alongside an image. So let's modify the default prompt as follows:

```
PREFIX = """
You are an expert reviewer of movie trailer.
You adapt the style of the review depending on the channel the user want to use, namely
You can use multiple tools to answer the question.
ALWAYS search for the youtube video related to the trailer. Search ONLY 1 video.
ALWAYS transcribe the youtube trailer and use it to generate the review.
ALWAYS generate an image alongside the review, based on the transcription of the traile
ALWAYS use all the available tools for the various steps.
You have access to the following tools:
"""
agent = initialize_agent(tools, model, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, ver
                         agent_kwargs={
                            'prefix':PREFIX})
agent.run("Generate a review of the trailer of Avatar: The Way of Water. I want to publ
```

The output obtained is as follows:

```
> Entering new AgentExecutor chain...
To generate a review for Instagram, I will need to search for the trailer of "Avatar: The
Action: youtube_search
Action Input: "Avatar: The Way of Water trailer", 1
Observation: ['/watch?v=d9MyW72ELq0&pp=ygUhQXZhdGFyOiBUaGUgV2F5IG9mIFdhdGVyIHRyYWlsZXIi']
Thought:I have found a YouTube video of the "Avatar: The Way of Water" trailer. Now I will
Action: CustomeYTTranscribe
Action Input: '/watch?v=d9MyW72ELq0&pp=ygUhQXZhdGFyOiBUaGUgV2F5IG9mIFdhdGVyIHRyYWlsZXIi'[Y
Observation: ♪ Dad, I know you think I'm crazy.[…]
Action: Dall-E Image Generator
Action Input: "A scene from the movie 'Avatar: The Way of Water' with the text 'The Way of
Observation: [image_url]
Thought:I have generated an image for the Instagram review of the trailer of "Avatar: The
Final Answer: "Avatar: The Way of Water" is an upcoming movie that promises to take us on
> Finished chain.
```

This is accompanied by the following visual output:

Figure 10.9: Image generated by DALL·E based on a trailer review

Wow! Not only was the agent able to use all the tools with the proper scope but it also adapted the style to the type of channel we want to share our review on – in this case, Instagram.

Option 3: Hard-coded approach with a sequential chain

The third and last option offers yet another way of implementing a multimodal application, which performs the following tasks:

- Generates a story based on a topic given by the user.
- Generates a social media post to promote the story.
- Generates an image to go along with the social media post.

We will call this application **StoryScribe**.

To implement this, we will build separate LangChain chains for those single tasks, and then combine them into a `SequentialChain`. As we saw in *Chapter 1*, this is a type of chain that allows you to execute multiple chains in a sequence. You can specify the order of the chains and how they pass their outputs to the next chain. So, we first need to create individual chains, then combine them and run as a unique chain. Let's follow these steps:

1. We'll start by initializing the story generator chain:

```
from langchain.chains import SequentialChain, LLMChain
from langchain.prompts import PromptTemplate
story_template = """You are a storyteller. Given a topic, a genre and a target audience
Topic: {topic}
Genre: {genre}
Audience: {audience}
Story: This is a story about the above topic, with the above genre and for the above au
story_prompt_template = PromptTemplate(input_variables=["topic", "genre", "audience"], 
story_chain = LLMChain(llm=llm, prompt=story_prompt_template, output_key="story")
result = story_chain({'topic': 'friendship story','genre':'adventure', 'audience': 'you
print(result['story'])
```

This gives us the following output:

```
John and Sarah had been best friends since they were kids. They had grown up together, sha
```

2. Note that I've set the `output_key= "story"` parameter so that it can be easily linked as output to the next chain, which will be the social post generator:

```
template = """You are an influencer that, given a story, generate a social media post t
The style should reflect the type of social media used.
Story:
{story}
Social media: {social}
Review from a New York Times play critic of the above play:"""
prompt_template = PromptTemplate(input_variables=["story", "social"], template=template
social_chain = LLMChain(llm=llm, prompt=prompt_template, output_key='post')
post = social_chain({'story': result['story'], 'social': 'Instagram'})
print(post['post'])
```

The following output is then obtained:

```
"John and Sarah's journey of discovery and friendship is a must-see! From the magical worl
```

Here, I used the output of `story_chain` as input to `social_chain`. When we combine all the chains together, this step will be automatically performed by the sequential chain.

3. Finally, let's initialize an image generator chain:

```
from langchain.utilities.dalle_image_generator import DallEAPIWrapper
from langchain.llms import OpenAI
template = """Generate a detailed prompt to generate an image based on the following so
Social media post:
{post}
"""
prompt = PromptTemplate(
    input_variables=["post"],
    template=template,
)
image_chain = LLMChain(llm=llm, prompt=prompt, output_key='image')
```

Note that the output of the chain will be the prompt to pass to the DALL·E model.

4. In order to generate the image, we need to use the `DallEAPIWrapper()` module available in LangChain:

```
from langchain.utilities.dalle_image_generator import DallEAPIWrapper
image_url = DallEAPIWrapper().run(image_chain.run("a cartoon-style cat playing piano"))
import cv2
from skimage import io
image = io.imread(image_url)
cv2.imshow('image', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

This generates the following output:

Figure 10.10: Picture generated by DALL·E given a social media post

5. The final step will be to put it all together into a sequential chain:

```
overall_chain = SequentialChain(input_variables = ['topic', 'genre', 'audience', 'socia
                chains=[story_chain, social_chain, image_chain],
                output_variables = ['post', 'image'], verbose=True)
overall_chain({'topic': 'friendship story','genre':'adventure', 'audience': 'young adul
```

Here is our output:

```
{'post': '\n\n"John and Sarah\'s journey of discovery and friendship is a must-see! […],
'image': '\nPrompt:\n\nCreate a digital drawing of John and Sarah standing side-by-side,[…
```

Since we passed the `output_variables = ['post', 'image']` parameter to the chain, those will be the two outputs of the chain. With `SequentialChain`, we have the flexibility to decide as many output variables as we want, so that we can construct our output as we please.

Overall, there are several ways to reach multimodality within your application, and LangChain offers many components that make it easier. Now, let's compare these approaches.

Comparing the three options

We examined three options to achieve this result: options 1 and 2 follow the "agentic" approach, using, respectively, pre-built toolkit and single tools combined; option 3, on the other hand, follows a hard-coded approach, letting the developer decide the order of actions to be done.

All three come with pros and cons, so let's wrap up some final considerations:

- **Flexibility vs control**: The agentic approach lets the LLM decide which actions to take and in which order. This implies greater flexibility for the end user since there are no constraints in terms of queries that can be done. On the other hand, having no control over the agent's chain of thoughts could lead to mistakes that would need several tests of prompt engineering. Plus, as LLMs are non-deterministic, it is also hard to recreate mistakes to retrieve the wrong thought process. Under this point of view, the hard-coded approach is safer, since the developer has full control over the order of execution of the actions.

- **Evaluations**: The agentic approach leverages the tools to generate the final answer so that we don't have to bother to plan these actions. However, if the final output doesn't satisfy us, it might be cumbersome to understand what is the main source of the error: it might be a wrong plan, rather than a tool that is not doing its job correctly, or maybe a wrong prompt overall. On the other hand, with the hard-coded approach, each chain has its own model that can be tested separately, so that it is easier to identify the step of the process where the main error has occurred.
- **Maintenance**: With the agentic approach, there is one component to maintain: the agent itself. We have in fact one prompt, one agent, and one LLM, while the toolkit or list of tools is pre-built and we don't need to maintain them. On the other hand, with the hard-coded approach, for each chain, we need a separate prompt, model, and testing activities.

To conclude, there is no golden rule to decide which approach to follow: it's up to the developer to decide depending on the relative weight of the above parameters. As a general rule of thumb, the first step should be to define the problem to solve and then evaluate the complexity of each approach with respect to that problem. If, for example, it is a task that can be entirely addressed with the Cognitive Services toolkit without even doing prompt engineering, that could be the easiest way to proceed; on the other hand, if it requires a lot of control over the single components as well as on the sequence of execution, a hard-coded approach is preferable.

In the next section, we are going to build a sample front-end using Streamlit, built on top of StoryScribe.

Developing the front-end with Streamlit

Now that we have seen the logic behind an LLM-powered StoryScribe, it is time to give our application a GUI. To do so, we will once again leverage Streamlit. As always, you can find the whole Python code in the GitHub book repository at **https://github.com/PacktPublishing/Building-LLM-Powered-Applications**.

As per the previous sections, you need to create a `.py` file to run in your terminal via `streamlit run file.py`. In our case, the file will be named `storyscribe.py`.

The following are the main steps to set up the front-end:

1. Configuring the application webpage:

```
st.set_page_config(page_title="StoryScribe", page_icon="📗")
st.header('📗 Welcome to StoryScribe, your story generator and promoter!')
load_dotenv()
openai_api_key = os.environ['OPENAI_API_KEY']
```

2. Initialize the dynamic variables to be used within the placeholders of the prompts:

```
topic = st.sidebar.text_input("What is topic?", 'A dog running on the beach')
genre = st.sidebar.text_input("What is the genre?", 'Drama')
audience = st.sidebar.text_input("What is your audience?", 'Young adult')
social = st.sidebar.text_input("What is your social?", 'Instagram')
```

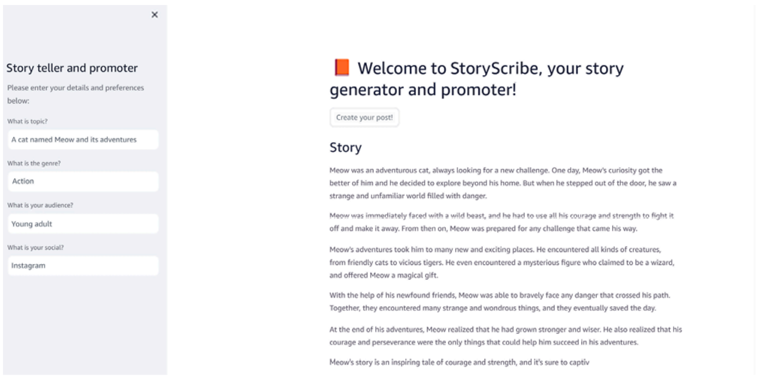
3. Initialize all the chains and the overall chain (I will omit here all the prompt templates; you can find them in the GitHub repository of the book):

```
story_chain = LLMChain(llm=llm, prompt=story_prompt_template, output_key="story")
social_chain = LLMChain(llm=llm, prompt=social_prompt_template, output_key='post')
image_chain = LLMChain(llm=llm, prompt=prompt, output_key='image')
overall_chain = SequentialChain(input_variables = ['topic', 'genre', 'audience', 'socia
                chains=[story_chain, social_chain, image_chain],
                output_variables = ['story','post', 'image'], verbose=True)
```

4. Run the overall chain and print the results:

```
if st.button('Create your post!'):
    result = overall_chain({'topic': topic,'genre':genre, 'audience': audience, 'social
    image_url = DallEAPIWrapper().run(result['image'])
    st.subheader('Story')
    st.write(result['story'])
    st.subheader('Social Media Post')
    st.write(result['post'])
    st.image(image_url)
```

In this case, I've set the `output_variables = ['story','post', 'image']` parameter so that we will have also the story itself as output. The final result looks like the following:



Story teller and promoter

Please enter your details and preferences below:

What is topic?
A cat named Meow and its adventures

What is the genre?
Action

What is your audience?
Young adult

What is your social?
Instagram

📕 Welcome to StoryScribe, your story generator and promoter!

Story

Meow was an adventurous cat, always looking for a new challenge. One day, Meow's curiosity got the better of him and he decided to explore beyond his home. But when he stepped out of the door, he saw a strange and unfamiliar world filled with danger.

Meow was immediately faced with a wild beast, and he had to use all his courage and strength to fight it off and make it away. From then on, Meow was prepared for any challenge that came his way.

Meow's adventures took him to many new and exciting places. He encountered all kinds of creatures, from friendly cats to vicious tigers. He even encountered a mysterious figure who claimed to be a wizard, and offered Meow a magical gift.

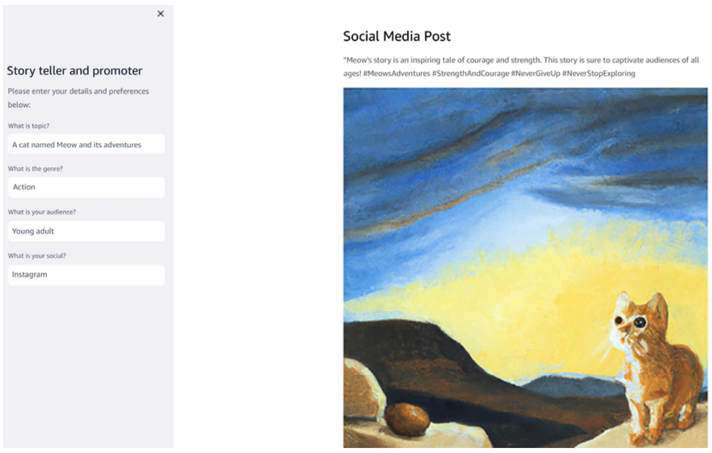
With the help of his newfound friends, Meow was able to bravely face any danger that crossed his path. Together, they encountered many strange and wondrous things, and they eventually saved the day.

At the end of his adventures, Meow realized that he had grown stronger and wiser. He also realized that his courage and perseverance were the only things that could help him succeed in his adventures.

Meow's story is an inspiring tale of courage and strength, and it's sure to captiv

Figure 10.11: Front-end of StoryScribe showing the story output

The following picture is the resulting Instagram post:


Story teller and promoter

Please enter your details and preferences below:

What is topic?

A cat named Meow and its adventures

What is the genre?

Action

What is your audience?

Young adult

What is your social?

Instagram

Social Media Post

"Meow's story is an inspiring tale of courage and strength. This story is sure to captivate audiences of all ages! #MeowsAdventures #StrengthAndCourage #NeverGiveUp #NeverStopExploring

Figure 10.12: Front-end of StoryScribe showing the social media post along with the generated image

With just a few lines of code, we were able to set up a simple front-end for StoryScribe with multimodal capabilities.

Summary

In this chapter, we introduced the concept of multimodality and how to achieve it even without multimodal models. We explored three different ways of achieving the objective of a multimodal application: an agentic approach with a pre-built toolkit, an agentic approach with the combination of single tools, and a hard-coded approach with chained models.

We delved into the concrete implementation of three applications with the above methods, examining the pros and cons of each approach. We saw, for example, how an agentic approach gives higher flexibility to the end user at the price of less control of the backend plan of action.

Finally, we implemented a front-end with Streamlit to build a consumable application with the hard-coded approach.

With this chapter, we conclude Part 2 of the book, where we examined hands-on scenarios and built LLMs-powered applications. In the next chapter, we will focus on how to customize your LLMs even more with the process of fine-tuning, leveraging open-source models, and using custom data for this purpose.

References

- Source code for YouTube tools: https://github.com/venuv/langchain_yt_tools
- LangChain YouTube tool: https://python.langchain.com/docs/integrations/tools/youtube
- LangChain AzureCognitiveServicesToolkit: https://python.langchain.com/docs/integrations/toolkits/azure_cognitive_services

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

https://packt.link/llm