

# Chapter 10. Building AI-powered Applications

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [ccollins@oreilly.com](mailto:ccollins@oreilly.com).

---

In this chapter you'll apply the five principles of prompting to an end-to-end AI workflow for content writing. The service will write blog posts based on the user's responses to interview questions, in the style of the user's writing. This system was first documented on the [Saxifrage blog](#).

## AI Blog Writing

The naive approach to creating a blog writing service using AI would be to prompt ChatGPT with `Write a blog post on {blogPostTopic}`. The resulting content would be of reasonable quality, but wouldn't likely contain any valuable opinions or unique experiences on the topic. The content would also likely be short and generic, and therefore unlikely to rank on Google.

A more sophisticated approach might be to build up a longer prompt with further instructions. Detail on the prescribed writing tone, architecture of the blog post, and keywords to include could be added. An example of a common blog post [writing prompt](#) can be seen here:

Input:

```
Create a blog post about "{blogPostTopic}". Write it in a "{tone}" tone.  
Use transition words.  
Use active voice. Write over 1000 words.  
Use very creative titles for the blog post.  
Add a title for each section. Ensure there are a minimum of 9 sections. Each  
section should have a minimum of two paragraphs.  
Include the following keywords: "{keywords}."  
Create a good slug for this post and a meta description with a maximum of 100  
words and add it to the end of the blog post.
```

This longer, more sophisticated prompt is likely to result in better quality content. However, let's run through the five principles of prompting as a checklist:

- *Direction*: There are some instructions provided, such as the tone, using transition words, and an active voice. However, the content is still likely to sound like AI, and not like the user.
- *Format*: Although there are some mentions of structure, including dictating nine sections of two paragraphs, it's likely these instructions will be ignored. ChatGPT is bad at math, and is often unable to follow instructions dictating a number of sections or words.
- *Examples*: There are no samples of how to do the task given, which is likely to harm the reliability of running this prompt across multiple topics or even multiple times on the same topic. Even providing one example (a 1-shot prompt) could radically help improve quality.
- *Evaluation*: This is an example of *blind prompting* (adding instructions to a prompt without testing them). It's likely some of these instructions make no difference to quality (unnecessarily costing tokens), or might even degrade quality.
- *Division*: The entire task is attempted with just one prompt, which is likely to harm performance. Without breaking the task into sub-tasks, it's hard to understand which part of the process is succeeding or failing.

Through this chapter, you'll create multiple LLM chain components. Each chain will be implemented in LangChain, in order to make it more maintainable, and to give easy logging for monitoring and optimization. The resulting system will help you generate *human-sounding* content based on the unique opinions and experiences of the user.

It's crucial that you first prepare your workspace with the necessary tools. Therefore, let's shift our focus towards topic research and start setting up your programming environment.

## Topic Research

You will need to install several Python packages to effectively use LangChain's document loaders, including the following:

- `google-searchresults`: A Python library designed to scrape and process Google search results.
- `pandas`: Offers data structures and operations for manipulating numerical tables and time series data.
- `html2text`: This tool converts HTML from files or web pages into markdown (*.md*) files or text.
- `pytest-playwright`: This package enables end-to-end testing with Playwright.
- `chromadb`: ChromaDB is an open source vector database.
- `nest_asyncio`: Extends the Python standard `asyncio` to patch and render it compatible with Jupyter Notebooks.

Installation of these packages can be achieved easily with this command:

```
pip install google-searchresults pandas html2text pytest-playwright chromadb \
nest_asyncio --quiet
```

Additionally you'll be using LangChain's document loaders that require Playwright.

Type this command on your terminal: `playwright install`

Additionally you'll need to choose a `TOPIC` and set environment variables for both `SERPAPI_API_KEY` and `STABILITY_API_KEY`. If you're running the script without Jupyter Notebook then you won't need to use any of the `nest_asyncio` code.

```
from langchain_openai.chat_models import ChatOpenAI
from langchain.output_parsers import PydanticOutputParser
from langchain.text_splitter import RecursiveCharacterTextSplitter
import os
```

```
# Custom imports:
from content_collection import collect_serp_data_and_extract_text_from_webpage
from custom_summarize_chain import create_all_summaries, DocumentSummary

import nest_asyncio
nest_asyncio.apply()

# Constant variables:
TOPIC = "Neural networks"
os.environ["SERPAPI_API_KEY"] = ""
os.environ["STABILITY_API_KEY"] = ""
```

Next, you'll focus on summarizing web content efficiently:

```
# Extract content from webpages into LangChain documents:
text_documents = await \
collect_serp_data_and_extract_text_from_webpages(TOPIC)

# LLM, text splitter + parser:
llm = ChatOpenAI(temperature=0)
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=1500, chunk_overlap=400
)
parser = PydanticOutputParser(pydantic_object=DocumentSummary)

summaries = await create_all_summaries(text_documents,
parser,
llm,
text_splitter)
```

First, import the required tools and then fetch the web page content related to your `TOPIC`. After setting up your `ChatOpenAI` model, you'll utilize a `text_splitter` to manage text chunks. The splitter ensures no snippet is too long, while maintaining context with overlap. Then create the `PydanticOutputParser` to handle and structure the summaries. By feeding the extracted documents through a dedicated summarization function, the LLM produces concise summaries.

If would like to dive deeper into the `create_all_summaries` function check [`custom\_summarize\_chain.py`](#) in `content/chapter_10`.

Some key points to highlight are that you can *subclass* most classes within LangChain. For example, you can override the default `ChromiumLoader` to be asynchronous:

```

from langchain_community.document_loaders import AsyncHtmlLoader, \
AsyncChromiumLoader


class ChromiumLoader(AsyncChromiumLoader):
    async def load(self):
        raw_text = [await self.ascrape_playwright(url) for url in self.urls]
        # Return the raw documents:
        return [Document(page_content=text) for text in raw_text]

    async def get_html_content_from_urls(
        df: pd.DataFrame, number_of_urls: int = 3, url_column: str = "link"
    ) -> List[Document]:
        # Get the HTML content of the first 3 URLs:
        urls = df[url_column].values[:number_of_urls].tolist()

        # If there is only one URL, convert it to a list:
        if isinstance(urls, str):
            urls = [urls]

        # Check for empty URLs:
        urls = [url for url in urls if url != ""]

        # Check for duplicate URLs:
        urls = list(set(urls))

        # Throw error if no URLs are found:
        if len(urls) == 0:
            raise ValueError("No URLs found!")
        # loader = AsyncHtmlLoader(urls) # Faster but might not always work.
        loader = ChromiumLoader(urls)
        docs = await loader.load()
        return docs

    async def create_all_summaries(
        # ... commented out for brevity
    ) -> List[DocumentSummary]:
        # ... commented out for brevity

```

By subclassing `ChromiumLoader`, you can easily create a custom implementation to *asynchronously scrape content* from multiple URLs using the Chrome browser. `get_html_content_from_urls` fetches HTML content from a list of URLs, ensuring no duplicates and handling potential errors.

# Expert Interview

Now that you've successfully extracted the summaries from Google for the top three results, you'll conduct an interview with an LLM, generating relevant questions to make sure that your article has a unique perspective using an `InterviewChain` class:

```
from expert_interview_chain import InterviewChain
interview_chain = InterviewChain(topic=TOPIC, document_summaries=summaries)
interview_questions = interview_chain()

for question in interview_questions.questions:
    print(f"Answer the following question: {question.question}\n", flush=True)
    answer = input(f"Answer the following question: {question.question}\n")
    print('-----')
    question.answer = answer
```

1. *InterviewChain instantiation*: With your topic and obtained summaries in hand, create an instance of `InterviewChain`, tailoring it to your data's unique context.
2. *Generating questions*: By simply calling the `interview_chain`, you kickstart the process of generating a series of probing questions derived from your summaries.
3. *Interactive q&a session*: Dive into an engaging loop where each derived question is printed, prompting you for an answer with `input()`. Your response is then saved back to the Pydantic object.

---

## GIVE DIRECTION

Giving an LLM unique answers provides unique context, and this allows an LLM to generate richer, more nuanced responses, ensuring your article offers a fresh and in-depth perspective.

---

All of the code for `InterviewChain` is in [`expert\_interview\_chain.py`](#). It has two significant components:

- A custom `System message`: This prompt includes role prompting, the previously generated summaries, topic, and format instructions (for the output parser):

```
system_message = """You are a content SEO researcher. Previously you have summarized and extracted key points from SERP results. The insights gained will be used to do content research and we will compare the key points, insights and summaries across multiple articles. You are now going to interview a content expert. You will ask them questions about the following topic: {topic}.
```

You must follow the following rules:

- Return a list of questions that you would ask a content expert about the topic.
- You must ask at least and at most 5 questions.
- You are looking for information gain and unique insights that are not already covered in the {document\_summaries} information.
- You must ask questions that are open-ended and not yes/no questions.

{format\_instructions}

- *Output parsers*: Diving deeper into the class, you encounter the `PydanticOutputParser`. This parser actively structures the LLMs responses into parsable, Pydantic `InterviewQuestions` objects:

```
# Set up a parser + inject instructions into the prompt template:  
parser = PydanticOutputParser(pydantic_object=InterviewQuestions)
```

In essence, you're orchestrating a conversation with the AI, instructing it to conceive potent questions that amplify content insights, all the while making customization a breeze.

## Generate Outline

Including the previous interview and research, you can generate an outline for the post with `BlogOutlineGenerator`. The `TOPIC`, `question_answers`, and Google `summaries` are passed to provide additional context:

```
from article_outline_generation import BlogOutlineGenerator  
  
blog_outline_generator = BlogOutlineGenerator(topic=TOPIC,  
questions_and_answers=[item.dict() for item in interview_questions.questions])  
  
questions_and_answers = blog_outline_generator.questions_and_answers  
outline_result = blog_outline_generator.generate_outline(summaries)
```

Let's explore the `BlogOutlineGenerator` class in detail:

```
from typing import List, Any
from pydantic.v1 import BaseModel

class SubHeading(BaseModel):
    title: str # Each subheading should have a title.

class BlogOutline(BaseModel):
    title: str
    sub_headings: List[SubHeading] # An outline has many sub_headings

# Langchain libraries:
from langchain.prompts.chat import (ChatPromptTemplate,
SystemMessagePromptTemplate)
from langchain.output_parsers import PydanticOutputParser
from langchain_openai.chat_models import ChatOpenAI

# Custom types:
from custom_summarize_chain import DocumentSummary

class BlogOutlineGenerator:
    def __init__(self, topic: str, questions_and_answers: Any):
        self.topic = topic
        self.questions_and_answers = questions_and_answers

        # Create a prompt
        prompt_content = """
Based on my answers and the summary, generate an outline for a blog
article on {topic}.
topic: {topic}
document_summaries: {document_summaries}
---
Here is the interview which I answered:
{interview_questions_and_answers}
---
Output format: {format_instructions}
"""

        system_message_prompt =
SystemMessagePromptTemplate.from_template(prompt_content)

        self.chat_prompt = ChatPromptTemplate.from_messages(
[system_message_prompt])

        # Create an output parser
        self.parser = PydanticOutputParser(pydantic_object=BlogOutline)
```

```

# Set up the chain
self.outline_chain = self.chat_prompt | ChatOpenAI() | self.parser

def generate_outline(self, summaries: List[DocumentSummary]) -> Any:
    print("Generating the outline...\\n---")
    result = self.outline_chain.invoke(
        {"topic": self.topic,
         "document_summaries": [s.dict() for s in summaries],
         "interview_questions_and_answers": self.questions_and_answers,
         "format_instructions": self.parser.get_format_instructions(),
        }
    )
    print("Finished generating the outline!\\n---")
    return result

```

A `BlogOutline` Pydantic object is created which contains `title` and `sub_headings` keys. Also, the outline chain is set up using LangChain expression language (LCEL) which passes the prompt into the chat model, then finally into the output parser:

```

# Set up the chain:
self.outline_chain = self.chat_prompt | ChatOpenAI() | self.parser

```

By using a Pydantic output parser, the chain will return a `BlogOutline` Pydantic object that will be used in future chains.

## Text Generation

After obtaining a summary, interview questions, and a blog post outline, it's time to start generating the text. The `ContentGenerator` class integrates SEO expertise with several LLM techniques, which include the following:

- *Embeddings and retrieval:* It efficiently splits and vectorizes original web pages, storing them in the `chroma` database and retrieving relevant web page text while writing each section.
- *Custom memory:* While crafting each blog section, it uses memory to avoid repeating the same information, while also summarizing the conversation if it becomes too long.

- *Bespoke context:* The LLM has a mixture of information, including your previous interview insights, what has been said before, and snippets of relevant web page text from Google:

```
from article_generation import ContentGenerator

content_gen = ContentGenerator(
    topic=TOPIC, outline=outline_result,
    questions_and_answers=questions_and_answers)

# Vectorize and store the original webpages:
content_gen.split_and_vectorize_documents(text_documents)
# Create the blog post:
blog_post = content_gen.generate_blog_post()
```

All of the source code is within [\*article\\_generation.py\*](#), but let's specifically focus on three components that are key to this chain.

The `OnlyStoreAIMemory` class is a customized subclass of `ConversationSummaryBufferMemory`:

```
from typing import List, Dict, Any
from langchain.memory import ConversationSummaryBufferMemory

from langchain_core.messages import SystemMessage

class OnlyStoreAIMemory(ConversationSummaryBufferMemory):
    def save_context(self, inputs: Dict[str, Any],
                     outputs: Dict[str, str]) -> None:
        input_str, output_str = self._get_input_output(inputs, outputs)
        self.chat_memory.add_ai_message(output_str)
```

It's tailored to ensure that the chat messages memory remains concise and relevant by *exclusively storing AI-generated messages*.

This deliberate choice bypasses storing retrieved documents that are used within the generation step, preventing memory bloat. Furthermore, the memory mechanism ensures the AI remains aware of its prior writings, enabling it to offer condensed summaries if the accumulated context surpasses set limits.

The `generate_blog_post` function loops through all of the subheadings and tries to retrieve as many relevant documents while fitting in the cur-

rent context length:

```
def generate_blog_post(self) -> List[str]:
    blog_post = []
    print("Generating the blog post...\n---")
    for subheading in self.outline.sub_headings:
        k = 5 # Initialize k
        while k >= 0:
            try:
                relevant_documents = (self.chroma_db.as_retriever() \
                    .get_relevant_documents(subheading.title,
                    k=k))
                section_prompt = f"""
...prompt_excluded_for_brevity...
Section text:
"""

                result = self.blog_post_chain.predict(section_prompt)
                blog_post.append(result)
                break
            except Exception as e:
                print(f"An error occurred: {e}")
                k -= 1
        if k < 0:
            print(''All attempts to fetch relevant documents have
failed. Using an empty string for relevant_documents.
''')
            relevant_documents = ""
    print("Finished generating the blog post!\n---")
    return blog_post
```

This function, `generate_blog_post`, iterates over each subheading. It attempts to fetch up to five relevant documents. If there's an issue fetching the documents, it smartly decreases the number and tries again. If all attempts fail, it gracefully defaults to no documents.

Finally, the prompt for generating each section is very context rich:

```
section_prompt = f"""
You are currently writing the section: {subheading.title}
---

Here are the relevant documents for this section: {relevant_documents}.
If the relevant documents are not useful, you can ignore them.
You must never copy the relevant documents as this is plagiarism.

---

Here are the relevant insights that we gathered from our interview questions
and answers: {self.questions_and_answers}.
```

You must include these insights where possible as they are important and will help our content rank better.

---

You must follow the following principles:

- You must write the section: {subheading.title}
- Render the output in .md format
- Include relevant formats such as bullet points, numbered lists, etc.

---

Section text:

"""

The `section_prompt` elegantly sets the stage by announcing the specific section you're working on, using `{subheading.title}`. But it doesn't stop there. By feeding the LLM with `{relevant_documents}`, it offers background and depth, while explicitly cautioning against plagiarism.

Moreover, by including insights from your interview via

`{self.questions_and_answers}`, the prompt ensures that valuable information is front and center. Finally, it sets clear expectations the format, the inclusion of certain features, and the topic at hand. This makes the LLM not just a tool but an informed co-author, working diligently alongside you to create content.

## Writing Style

Now that the article is written, we can go a step further in terms of making it sound uniquely human, by rewriting the content in a specific writing style. This will go a long way in making the content less detectable as obviously AI (though ethically you should still declare any AI assistance), and decrease the amount of time you spend editing the final draft before publishing.

Before rewriting you need to know what writing style you want to emulate, be it your own or someone else's. One common approach is to ask ChatGPT to summarize the writing style of someone who is famous, or at least popular enough in your industry to appear in ChatGPT's training data. Commonly the model will want to respond with the name of the author and examples of writing, so adding instructions not to, and ending the prompt with a bullet point (or an `-` character in this case) gets the format you need:

Input:

Describe in detail the writing style of Harry Dry from MarketingExamples.com  
Do not mention the writer or source, and respond only with bullet points:

-

Output:

- Concise and to the point
- Humorous and knowledgeable tone
- Relatable and accessible language
- Uses storytelling to convey ideas
- Relies on examples and case studies
- Provides actionable advice and tips
- Uses subheadings and bullet points for easy readability
- Emphasizes key points or takeaways with bold or italicized text
- Addresses the reader directly, creating a conversational tone

Often provided only a single sample of text, ChatGPT can generate a reasonable writing style guide, which can then be used for rewriting. Once the writing style is defined, elements can be mixed and matched to arrive at a more ideal style. The following example takes elements of Mike Taylor's writing style from the previous example, and elements of Harry Dry's writing style from the example prior. This is another example of meme unbundling, as discussed in [Chapter 5](#):

Rewrite the article in the following style:

- Concise and to the point
- Professional and knowledgeable tone
- Relatable and accessible language
- Uses storytelling to convey ideas
- Relies on examples and case studies
- Mixes personal anecdotes with industry insights
- Provides actionable advice and tips
- Uses subheadings and bullet points for easy readability

In practice this part of the AI writing process is the most difficult to get right, and it's the only one that requires the larger and more expensive GPT-4 model to get passable results. If this part of the process isn't right, the user can be left doing a lot of manual editing to get the writing in the house style. Given the strategic importance of this prompt it makes sense to do a round of [prompt optimization](#), trying multiple approaches.

When optimizing prompts you can run the same prompt multiple times and check the average performance against an evaluation metric. As an example, here are the results of testing five different prompt approaches against an evaluation metric of embedding distance. The lower the score the closer the embeddings of the response were to a reference answer (the text as rewritten manually is in the correct style). The prompts tested were as follows:

- *A*: Control - the standard prompt as detailed in the preceding example.
- *B*: 1-shot writing sample - we provided one sample of text, and asked GPT-4 to describe the writing style.
- *C*: 3-shot rewriting example - we gave three samples of the input text to GPT-4 and the rewritten version and asked it to describe the writing style.
- *D*: 3-shot writing sample - same as above, except without the input text, only the final samples of Mike's writing.

These prompts were [tested in an experiment we ran](#) against three test cases – `memetics`, `skyscraper technique`, and `value-based pricing` – which were snippets of text that were first generated by ChatGPT on a topic, for example: `explain value-based pricing`. We then manually rewrote the text in the style we desired to make reference texts for comparison. The embedding distance was calculated by getting the embeddings for the reference text (from OpenAI's `text-embedding-ada-002`) and comparing them to the embeddings for the output from the prompt, using *cosine similarity* (a method for calculating the distance between two sets of numbers), as detailed in [Langchain's embedding evaluator \(Figure 10-1\)](#).

AVERAGE of embedding_distance case					
variation	Mematics	SkyScrapers Technique	Value-Based Pricing	Grand Total	+/-
A: Control	0.053	0.042	0.061	0.052	0.00%
B: 1-Shot Writing Sample	0.045	0.034	0.048	0.043	-17.75%
C: 3-Shot Rewriting Example	0.049	0.046	0.058	0.051	-1.71%
D: 3-Shot Writing Sample	0.044	0.037	0.052	0.045	-14.03%
<b>Grand Total</b>	<b>0.048</b>	<b>0.040</b>	<b>0.055</b>	<b>0.047</b>	

Figure 10-1. Test results from prompt optimization

As you can see from the results in [Figure 10-1](#), some prompts work better than others, and some cases are easier for the AI to deliver on. It's important to test across multiple cases, with 10 or more runs per case, to get a realistic result for each prompt. Otherwise the non-deterministic nature of the responses might mean you'll think the performance was better or

worse than you can actually expect when scaling up usage of a prompt.

Here was the final resulting prompt that performed best:

You will be provided with the sample text.

Your task is to rewrite the text into a different writing style.

The writing style can be described as follows:

1. **Informative and Analytical:** The writer presents detailed information about different strategies, especially the main theme of the text, and breaks down its benefits, challenges, and implementation steps. This depth of information shows that the writer has a solid grasp of the topic.
2. **Structured and Organized:** The writing follows a logical flow, starting with a brief overview of different approaches, delving into a deep dive on the topic, and concluding with potential challenges and contexts where it might be best applied.
3. **Conversational Tone with Professionalism:** While the information is presented in a professional manner, the writer uses a conversational tone ("Here's how to implement..."), which makes it more relatable and easier for readers to understand.
4. **Practical and Actionable:** The writer not only explains the concept but also offers actionable advice ("Here's how to implement X") with step-by-step guidance based on real world experience.
5. **Balanced Perspective:** The writer doesn't just present the benefits of the topic but also discusses its challenges, which gives a well-rounded perspective to readers.
6. **Examples and Analogies:** To make concepts clearer, the writer uses concrete examples (e.g., how much a company might save per month) and analogies (e.g., making comparisons to popular frames of reference). This helps readers relate to the concepts and understand them better.
7. **Direct and Clear:** The writer uses straightforward language without excessive jargon. Concepts are broken down into digestible bits, making it accessible for a broad audience, even if they're not well-versed in business strategies. In essence, this writing style is a blend of professional analysis with practical, actionable advice, written in a clear and conversational tone.

---

#### EVALUATE QUALITY

Without testing the writing style, it would be hard to guess which prompting strategy would win. With a small amount of testing, you can be more confident this is the correct approach. Testing doesn't have to be highly organized or systematized, and the builders of many successful AI products like [GitHub CoPilot](#) admit their eval process was haphazard and messy (but it got the job done!).

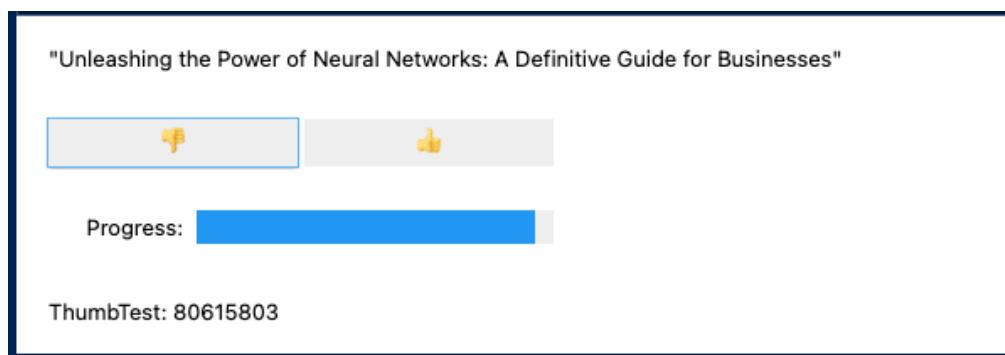
---

In this project we'll use this well-tested example, but you may take this opportunity to try and beat this score. The repository with the reference

texts and code is [publicly available on GitHub](#), and please feel free to contribute to the repository if you find a better approach. One potential path to try is fine-tuning, which may get you better results in matching the writing style if you have enough samples ([OpenAI recommends at least 50](#)). Even if you don't perform an A/B test (comparing two versions of a prompt to see which one performs better) on this prompt, hopefully these results convince you of the value of testing your prompts in general.

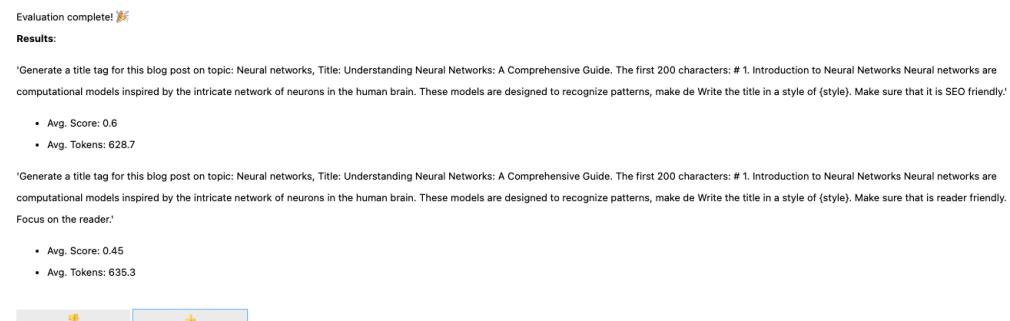
## Title Optimization

You can optimize the content's title by generating various options, testing them through A/B prompts and gauging their effectiveness with a thumbs up/down rating system as shown in [Figure 10-2](#).



*Figure 10-2. A simple thumbs up and thumbs down rating system*

After evaluating all the prompts, you'll be able to see which prompt had the highest average score and the token usage ([Figure 10-3](#)).



*Figure 10-3. Example A/B test results after manually evaluating a prompt*

## AI Blog Images

One thing you can do to make your blog look more professional is to add custom illustrations to your blog posts, with a consistent style. At its maxi-

mum this may mean training a Dreambooth model, as covered in [Chapter 9](#), on your brand styleguide or a mood board of images with a certain visual consistency or aesthetic quality you value. In many cases, however, training a custom model is not necessary, because a style can be replicated well using simple prompting.

One popular visual style among business-to-business (B2B) companies, [Corporate Memphis](#), is characterised by its vibrant color palettes, bold and asymmetric shapes, and a mix of both organic and geometric forms. This style arose as a [costly signaling technique](#), showing that the company could afford to commission custom illustrations from a designer, and therefore was serious enough to be trusted. You can replicate this style with AI, saving yourself the cost of custom illustrations, while benefiting from the prior associations formed in consumers' minds. [Figure 10-4](#) shows an example of Corporate Memphis style generated by Stable Diffusion, via the Stability AI API.

Input:

```
illustration of websites being linked together. in the style of corporate memphis  
white background, professional, clean lines, warm pastel colors
```

The output is shown in [Figure 10-4](#).



Figure 10-4. Corporate Memphis: “websites being linked together”

---

#### GIVE DIRECTION

Stable Diffusion is trained on many different styles, including obscure or niche styles like Corporate Memphis. If you know the name of a style, often that's all that's needed to guide the model towards the desired image. You can find a variety of art styles within this [visual prompt builder](#).

---

In our blog writing project we could ask the user for an idea of what image they want to accompany the blog post, but let's make it easier for them and automate this step. You can make an API call to ChatGPT and get back an idea for what could go in the image. When you get that response it can form the basis of your prompt to Stability AI, a technique called *meta-prompts*, where one AI model writes the prompt for another AI model.

Input:

Describe an image that would go well at the top of this article:

```
{text}
```

Output:

```
A seamless collage or mosaic of diverse cultural elements from around the world including traditional dances, art pieces, landmarks, and people in various traditional attires, symbolizing the interconnectedness of human cultures.
```

Stability AI hosts Stable Diffusion, including the latest models like Stable Diffusion XL, in their DreamStudio platform. You can also call them [via API](#), or via the Stability AI SDK (a library that simplifies the process of making the API call). In the following example we'll create a function for calling Stability AI with our prompt.

Input:

```
import base64
import os
import requests
import uuid

engine_id = "stable-diffusion-xl-1024-v1-0"
api_host = os.getenv('API_HOST', 'https://api.stability.ai')
api_key = os.getenv("STABILITY_API_KEY")

def generate_image(prompt):
    response = requests.post(
        f"{api_host}/v1/generation/{engine_id}/text-to-image",
        headers={
            "Content-Type": "application/json",
            "Accept": "application/json",
            "Authorization": f"Bearer {api_key}"
        },
        json={
            "text_prompts": [
                {
                    "text": '''an illustration of "+prompt+". in the style of
corporate memphis,
white background, professional, clean lines, warm pastel
colors'''
                }
            ],
            "cfg_scale": 7,
            "height": 1024,
```

```
        "width": 1024,
        "samples": 1,
        "steps": 30,
    },
)

if response.status_code != 200:
    raise Exception("Non-200 response: " + str(response.text))

data = response.json()

image_paths = [ ]

for i, image in enumerate(data["artifacts"]):
    filename = f"uuid.uuid4().hex[:7].png"
    with open(filename, "wb") as f:
        f.write(base64.b64decode(image["base64"]))

    image_paths.append(filename)

return image_paths

prompt = """A seamless collage or mosaic of diverse cultural elements from around the world, including traditional dances, art pieces, landmarks, and people in various traditional attires, symbolizing the interconnectedness of human cultures."""

generate_image(prompt)
```

The output is shown in [Figure 10-5](#).



Figure 10-5. A seamless collage or mosaic of diverse cultural elements from around the world

In order to encapsulate the whole system for image generation, you can bring the call to ChatGPT and the resulting call to Stability AI together in one function that uses the `outline_result.title`:

```
from image_generation_chain import create_image
image = create_image(outline_result.title)
```

The `create_image` function in [`image\_generation\_chain.py`](#) utilizes Stable diffusion to create an image based on a generated title from GPT-4:

```
import base64
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.messages import SystemMessage
import os
import requests
import uuid

engine_id = "stable-diffusion-xl-1024-v1-0"
api_host = os.getenv("API_HOST", "https://api.stability.ai")
```

```
api_key = os.getenv("STABILITY_API_KEY", "INSERT_YOUR_IMAGE_API_KEY_HERE")

if api_key == "INSERT_YOUR_IMAGE_API_KEY_HERE":
    raise Exception(
        '''You need to insert your API key in the
        image_generation_chain.py file.'''
        "You can get your API key from https://platform.openai.com/"
    )

def create_image(title) -> str:
    chat = ChatOpenAI()
    # 1. Generate the image prompt:
    image_prompt = chat.invoke(
        [
            SystemMessage(content=f"""Create an image prompt
                that will be used for MidJourney for {title}."""
        )
    ]
).content

# 2. Generate the image::
response = requests.post(
    f"{api_host}/v1/generation/{engine_id}/text-to-image",
    headers={
        "Content-Type": "application/json",
        "Accept": "application/json",
        "Authorization": f"Bearer {api_key}",
    },
    json={
        "text_prompts": [
            {
                "text": f'''an illustration of {image_prompt} in the
                    style of corporate memphis, white background,
                    professional, clean lines, warm pastel colors'''
            }
        ],
        "cfg_scale": 7,
        "height": 1024,
        "width": 1024,
        "samples": 1,
        "steps": 30,
    },
)
if response.status_code != 200:
    raise Exception("Non-200 response: " + str(response.text))
```

```
data = response.json()
image_paths = []

for i, image in enumerate(data["artifacts"]):
    filename = f"{uuid.uuid4().hex[:7]}.png"
    with open(filename, "wb") as f:
        f.write(base64.b64decode(image["base64"]))
    image_paths.append(filename)
return image_paths
```

Here's the high-level process:

1. With the `ChatOpenAI` model, you'll craft an image prompt for your given `title`.
2. Using the Stability AI API, you'll send this prompt to generate an image with precise styling instructions.
3. Then you'll decode and save this image locally using a unique filename and return its path.

With these steps, you're not just prompting the AI to create textual content, but you're directing it to bring your prompts to life visually.

This system is flexible based on whatever style you decide to use for blog images. Parameters can be adjusted as needed, and perhaps this API call can be replaced in future with a call to a custom fine-tuned Dreambooth model of your own. In the meantime, however, you have a quick and easy way to generate a custom image for each blog post, without requiring any further input from the user, in a consistent visual style.

## User Interface

Now that you have your script working end to end, you probably want to make it a little easier to work with, and maybe even get it into the hands of people who can give you feedback. The frontend of many AI tools in production is typically built using JavaScript, specifically the [NextJS](#) framework based on React. This is usually paired with a CSS library such as [TailwindCSS](#), which makes rapid prototyping of design elements easier.

However, most of your AI code is likely in Python at this stage, and switching programming languages and development environments can be a daunting challenge. As well as learning JavaScript, NextJS, and Tailwind, you may also run into a series of issues getting a server running

for your Python code, a database live for your application and user data, and then integrating all of that with a frontend web design.

Instead of spending a lot of time spinning up servers, building databases, and adjusting button colors, it might make sense to create a simple prototype frontend to get early feedback, before investing too much at this stage in an unproven idea. Once you have built and tested a simple interface, you'll have a better understanding of what to build when you do need to get your app production-ready.

For launching simple user interfaces for AI-based prototypes, there are several popular open-source interfaces, including [Gradio](#) and [Streamlit](#). Gradio was acquired by HuggingFace, and powers the web user interface for many interactive demos of open-source AI models, famously including the [AUTOMATIC1111](#) Stable Diffusion Web UI. You can quickly build a Gradio interface to make it easier to run your code locally, as well as sharing the prototype to get feedback.

We've created an interface that allows you to automate the entire process within two steps. You can get access to the [Gradio source code here](#).

Then run the Gradio application by going into the *chapter\_10* folder within your terminal and running: `python3 gradio_code_example.py`. The script will ask you to enter a `SERPAPI_API_KEY` and `STABILITY_API_KEY` in your terminal.

Then you can access the Gradio interface as shown in [Figure 10-6](#).

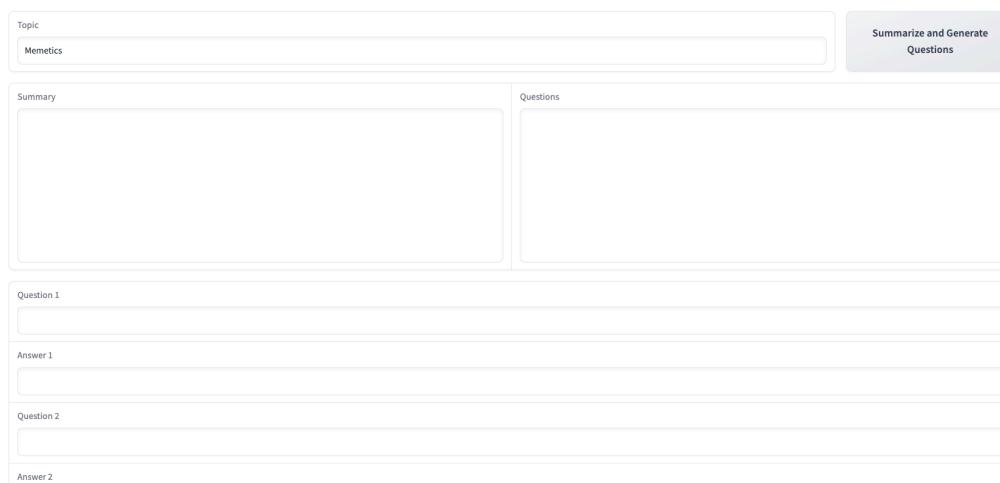


Figure 10-6. Gradio user interface

When you run Gradio you get an inline interface you can use directly, or a URL that you can click to open the web interface in your browser. If you

run Gradio with the parameter `share=True`, for example  
`demo.launch(share=True)` you get a publicly accessible link to share with friends, coworkers, or early users to get feedback on your prototype.

After initializing the interface, input a topic by clicking the Summarize and Generate Questions button. This will then collect and summarize the Google results as well as generate interview questions.

You'll then need to fill in the answers for each question. Finally, click the Generate Blog Post & Image button, which will take all the questions, summaries and will create an entire blog post and image using GPT-4!

---

#### EVALUATE QUALITY

The most valuable evaluation data in AI is human feedback, as it has been the key to many AI alignment breakthroughs, including those that power ChatGPT. Asking for feedback from users via a user interface, or even building feedback mechanisms into your product, helps you identify and fix edge cases.

---

If you are building for research purposes or wish to contribute to the open-source community, consider sharing your Gradio demo on HuggingFace spaces. Hugging Face Spaces allows anyone to host their Gradio demos freely, and uploading your project only takes a few minutes. New spaces can be created via the [HuggingFace website](#), or done programmatically using the HuggingFace API.

## Summary

Congratulations! You've journeyed through the comprehensive world of prompt engineering for generative AI. You started with learning the prompt engineering principles and explored the historical context of LLMs, gaining awareness of their capabilities and the privacy concerns they pose.

You learned how to extract structured data, apply best practices of prompt engineering, and familiarize yourself with an LLM package called LangChain. Then you discovered vector databases for storing and querying text based on similarity and ventured into the world of autonomous agents.

Also, you immersed yourself in image generation techniques using diffusion models, learning how to navigate through this latent space. Your journey covered everything from format modifiers and art style replication to Inpainting and Outpainting techniques. Moreover, you delved into more advanced usage cases such as prompt expansion, meme mapping, and CLIP Interrogator, alongside many others.

Finally, you transitioned towards utilizing prompt engineering for content writing. You learned about creating a blog writing service that generates posts based on user responses, mimicking their writing styles, along with topic research strategies.

Overall this journey not only enriched your knowledge but also equipped you with practical skills, setting you up to work professionally in the field of prompt engineering.

It's been our pleasure to guide you through the wide domain of prompt engineering for generative AI. Thank you for staying with us to the end of this book. We trust it will become a useful tool in all your future work with AI.

We would also greatly appreciate hearing your thoughts about the book, as well as any remarkable projects you create using the techniques we've discussed.

Please feel free to share your feedback or showcase your work by emailing us at [hi@brightpool.dev](mailto:hi@brightpool.dev). Once again, thank you! Your curiosity and perseverance is what shapes the future of this exciting field, and we can't wait to see what you contribute.

Happy prompting!