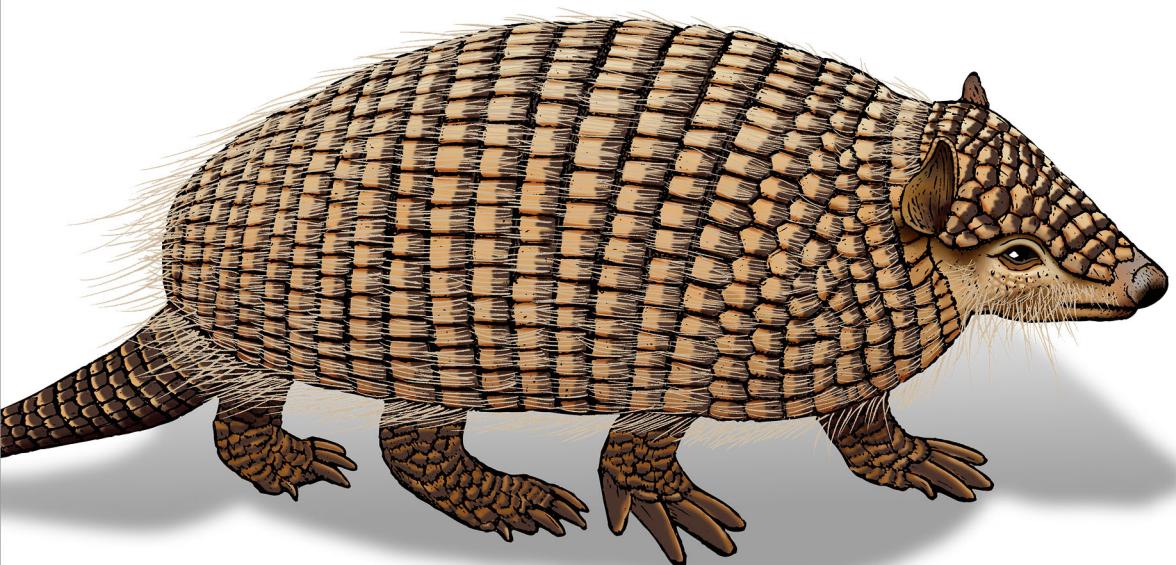


O'REILLY®

# Prompt Engineering for Generative AI

Future-Proof Inputs for Reliable AI Outputs



James Phoenix  
& Mike Taylor

# Prompt Engineering for Generative AI

Large language models (LLMs) and diffusion models such as ChatGPT and DALL-E have unprecedented potential. Having been trained on public text and images on the internet, they can contribute to a wide variety of tasks. And with the barrier to entry greatly reduced, practically any developer can harness AI models to tackle problems previously unsuitable for automation.

With this book, you'll gain a solid foundation in generative AI and learn how to apply these models in practice. When integrating LLMs and diffusion models into their workflows, most developers struggle to coax out reliable results for use in automated systems. Authors James Phoenix and Mike Taylor show you how prompt engineering principles will enable you to work effectively with AI in production.

This book explains:

- The five principles of prompting that are transferable across models, and will continue to work in the future
- Applying generative AI to real-world examples using libraries and frameworks such as LangChain
- Evaluating OpenAI models such as GPT-4 and DALL-E 2 against alternatives, including open-source models, comparing strengths and weaknesses
- How these principles apply in practice in the domains of NLP, text and image generation, and code

"The absolute best book-length resource I've read on prompt engineering. Mike and James are masters of their craft."

—Dan Shipper  
Cofounder & CEO, Every

"If you're looking to improve the accuracy and reliability of your AI systems, this book should be on your shelf."

—Mayo Oshin  
Founder & CEO, Siennai Analytics,  
early LangChain contributor

---

DATA

US \$79.99 CAN \$99.99

ISBN: 978-1-098-15343-4



9 781098 153434

[linkedin.com/company/oreilly-media](https://www.linkedin.com/company/oreilly-media)  
[youtube.com/oreillymedia](https://www.youtube.com/oreillymedia)

## Praise for *Prompt Engineering for Generative AI*

The absolute best book-length resource I've read on prompt engineering.

Mike and James are masters of their craft.

*—Dan Shipper, cofounder and CEO, Every*

This book is a solid introduction to the fundamentals of prompt engineering and generative AI. The authors cover a wide range of useful techniques for all skill levels from beginner to advanced in a simple, practical, and easy-to-understand way. If you're looking to improve the accuracy and reliability of your AI systems, this book should be on your shelf.

*—Mayo Oshin, founder and CEO, Siennai Analytics,  
early LangChain contributor*

Phoenix and Taylor's guide is a lighthouse amidst the vast ocean of generative AI. Their book became a cornerstone for my team at Phiture AI Labs, as we learned to harness LLMs and diffusion models for creating marketing assets that resonate with the essence of our clients' apps and games. Through prompt engineering, we've been able to generate bespoke, on-brand content at scale. This isn't just theory; it's a practical masterclass in transforming AI's raw potential into tailored solutions, making it an essential read for developers looking to elevate their AI integration to new heights of creativity and efficiency.

*—Moritz Daan, Founder/Partner,  
Phiture Mobile Growth Consultancy*

*Prompt Engineering for Generative AI* is probably the most future-proof way of future-proofing your tech career. This is without a doubt the best resource for anyone working in practical applications of AI. The rich, refined principles in here will help both new and seasoned AI engineers stay on top of this very competitive game for the foreseeable future.

—Ellis Crosby, CTO and cofounder, Incremento

This is an essential guide for agency and service professionals. Integrating AI with service and client delivery, using automation management, and speeding up solutions will set new industry standards. You'll find useful, practical information and tactics in the book, allowing you to understand and utilize AI to its full potential.

—Byron Tassoni-Resch, CEO and cofounder, WeDiscover

A really interesting and informative read, mixing practical tips and tricks with some solid foundational information. The world of GenAI is developing at breakneck speed, and having a toolset that can deliver results, regardless of the foundational model being used, is worth its weight in gold!

—Riaan Dreyer, chief digital and data officer, Bank of Iceland

The authors expertly translate prompt engineering intricacies into a practical toolkit for text and image generation. This guide, spanning standard practices to cutting-edge techniques, empowers readers with practical tips to maximize generative AI model capabilities.

—Aditya Goel, generative AI consultant

---

# Prompt Engineering for Generative AI

*Future-Proof Inputs for Reliable  
AI Outputs at Scale*

*James Phoenix and Mike Taylor*

O'REILLY®

## **Prompt Engineering for Generative AI**

by James Phoenix and Mike Taylor

Copyright © 2024 Saxifrage, LLC and Just Understanding Data LTD. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Acquisitions Editor:** Nicole Butterfield

**Indexer:** nSight, Inc.

**Development Editor:** Corbin Collins

**Interior Designer:** David Futato

**Copyeditor:** Piper Editorial Consulting, LLC

**Cover Designer:** Karen Montgomery

**Proofreader:** Kim Wimpsett

**Illustrator:** Kate Dullea

May 2024: First Edition

### **Revision History for the First Edition**

2024-05-15: First Release

2024-10-18: Second Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098153434> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Prompt Engineering for Generative AI*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-15343-4

[LSI]

---

# Table of Contents

Preface.....	xı
<b>1. The Five Principles of Prompting.....</b>	<b>1</b>
Overview of the Five Principles of Prompting	4
1. Give Direction	8
2. Specify Format	14
3. Provide Examples	17
4. Evaluate Quality	20
5. Divide Labor	32
Summary	38
<b>2. Introduction to Large Language Models for Text Generation.....</b>	<b>41</b>
What Are Text Generation Models?	41
Vector Representations: The Numerical Essence of Language	42
Transformer Architecture: Orchestrating Contextual Relationships	43
Probabilistic Text Generation: The Decision Mechanism	45
Historical Underpinnings: The Rise of Transformer Architectures	46
OpenAI's Generative Pretrained Transformers	48
GPT-3.5-turbo and ChatGPT	48
GPT-4	51
Google's Gemini	51
Meta's Llama and Open Source	52
Leveraging Quantization and LoRA	53
Mistral	53
Anthropic: Claude	54
GPT-4V(ision)	54
Model Comparison	54
Summary	55

<b>3. Standard Practices for Text Generation with ChatGPT.....</b>	<b>57</b>
Generating Lists	57
Hierarchical List Generation	59
When to Avoid Using Regular Expressions	64
Generating JSON	64
YAML	66
Filtering YAML Payloads	67
Handling Invalid Payloads in YAML	69
Diverse Format Generation with ChatGPT	72
Mock CSV Data	73
Explain It like I'm Five	74
Universal Translation Through LLMs	74
Ask for Context	76
Text Style Unbundling	80
Identifying the Desired Textual Features	80
Generating New Content with the Extracted Features	82
Extracting Specific Textual Features with LLMs	82
Summarization	82
Summarizing Given Context Window Limitations	83
Chunking Text	85
Benefits of Chunking Text	86
Scenarios for Chunking Text	86
Poor Chunking Example	87
Chunking Strategies	89
Sentence Detection Using SpaCy	90
Building a Simple Chunking Algorithm in Python	91
Sliding Window Chunking	92
Text Chunking Packages	94
Text Chunking with Tiktok	95
Encodings	95
Understanding the Tokenization of Strings	95
Estimating Token Usage for Chat API Calls	97
Sentiment Analysis	99
Techniques for Improving Sentiment Analysis	101
Limitations and Challenges in Sentiment Analysis	101
Least to Most	102
Planning the Architecture	102
Coding Individual Functions	102
Adding Tests	103
Benefits of the Least to Most Technique	104
Challenges with the Least to Most Technique	105
Role Prompting	105

Benefits of Role Prompting	106
Challenges of Role Prompting	106
When to Use Role Prompting	107
GPT Prompting Tactics	107
Avoiding Hallucinations with Reference	108
Give GPTs “Thinking Time”	110
The Inner Monologue Tactic	111
Self-Eval LLM Responses	112
Classification with LLMs	114
Building a Classification Model	115
Majority Vote for Classification	116
Criteria Evaluation	117
Meta Prompting	120
Summary	124
<b>4. Advanced Techniques for Text Generation with LangChain.....</b>	<b>125</b>
Introduction to LangChain	125
Environment Setup	127
Chat Models	128
Streaming Chat Models	130
Creating Multiple LLM Generations	130
LangChain Prompt Templates	131
LangChain Expression Language (LCEL)	132
Using PromptTemplate with Chat Models	134
Output Parsers	134
LangChain Evals	138
OpenAI Function Calling	146
Parallel Function Calling	149
Function Calling in LangChain	151
Extracting Data with LangChain	152
Query Planning	153
Creating Few-Shot Prompt Templates	154
Fixed-Length Few-Shot Examples	155
Formatting the Examples	155
Selecting Few-Shot Examples by Length	157
Limitations with Few-Shot Examples	159
Saving and Loading LLM Prompts	159
Data Connection	161
Document Loaders	162
Text Splitters	165
Text Splitting by Length and Token Size	165
Text Splitting with Recursive Character Splitting	167

Task Decomposition	169
Prompt Chaining	171
Sequential Chain	172
itemgetter and Dictionary Key Extraction	173
Structuring LCEL Chains	178
Document Chains	179
Stuff	181
Refine	182
Map Reduce	182
Map Re-rank	183
Summary	184
<b>5. Vector Databases with FAISS and Pinecone.....</b>	<b>185</b>
Retrieval Augmented Generation (RAG)	188
Introducing Embeddings	189
Document Loading	198
Memory Retrieval with FAISS	201
RAG with LangChain	206
Hosted Vector Databases with Pinecone	207
Self-Querying	215
Alternative Retrieval Mechanisms	220
Summary	221
<b>6. Autonomous Agents with Memory and Tools.....</b>	<b>223</b>
Chain-of-Thought	223
Agents	225
Reason and Act (ReAct)	227
Reason and Act Implementation	229
Using Tools	235
Using LLMs as an API (OpenAI Functions)	237
Comparing OpenAI Functions and ReAct	241
Use Cases for OpenAI Functions	242
ReAct	242
Use Cases for ReAct	243
Agent Toolkits	243
Customizing Standard Agents	245
Custom Agents in LCEL	246
Understanding and Using Memory	249
Long-Term Memory	249
Short-Term Memory	249
Short-Term Memory in QA Conversation Agents	250
Memory in LangChain	250

Preserving the State	251
Querying the State	252
ConversationBufferMemory	252
Other Popular Memory Types in LangChain	255
ConversationBufferWindowMemory	255
ConversationSummaryMemory	255
ConversationSummaryBufferMemory	256
ConversationTokenBufferMemory	256
OpenAI Functions Agent with Memory	257
Advanced Agent Frameworks	259
Plan-and-Execute Agents	259
Tree of Thoughts	260
Callbacks	261
Global (Constructor) Callbacks	263
Request-Specific Callbacks	264
The Verbose Argument	264
When to Use Which?	264
Token Counting with LangChain	264
Summary	266
<b>7. Introduction to Diffusion Models for Image Generation.....</b>	<b>267</b>
OpenAI DALL-E	271
Midjourney	273
Stable Diffusion	277
Google Gemini	279
Text to Video	279
Model Comparison	279
Summary	280
<b>8. Standard Practices for Image Generation with Midjourney.....</b>	<b>283</b>
Format Modifiers	283
Art Style Modifiers	287
Reverse Engineering Prompts	289
Quality Boosters	290
Negative Prompts	292
Weighted Terms	294
Prompting with an Image	297
Inpainting	300
Outpainting	303
Consistent Characters	305
Prompt Rewriting	307
Meme Unbundling	310

Meme Mapping	315
Prompt Analysis	317
Summary	318
<b>9. Advanced Techniques for Image Generation with Stable Diffusion.....</b>	<b>319</b>
Running Stable Diffusion	319
AUTOMATIC1111 Web User Interface	326
Img2Img	334
Upscaling Images	337
Interrogate CLIP	339
SD Inpainting and Outpainting	340
ControlNet	343
Segment Anything Model (SAM)	352
DreamBooth Fine-Tuning	355
Stable Diffusion XL Refiner	362
Summary	365
<b>10. Building AI-Powered Applications.....</b>	<b>367</b>
AI Blog Writing	367
Topic Research	368
Expert Interview	371
Generate Outline	373
Text Generation	374
Writing Style	377
Title Optimization	380
AI Blog Images	381
User Interface	387
Summary	389
<b>Index.....</b>	<b>391</b>

---

# Preface

The rapid pace of innovation in generative AI promises to change how we live and work, but it's getting increasingly difficult to keep up. The number of [AI papers published on arXiv is growing exponentially](#), [Stable Diffusion](#) has been among the fastest growing open source projects in history, and AI art tool [Midjourney's Discord server](#) has tens of millions of members, surpassing even the largest gaming communities. What most captured the public's imagination was OpenAI's release of ChatGPT, [which reached 100 million users in two months](#), making it the fastest-growing consumer app in history. Learning to work with AI has quickly become one of the most in-demand skills.

Everyone using AI professionally quickly learns that the quality of the output depends heavily on what you provide as input. The discipline of *prompt engineering* has arisen as a set of best practices for improving the reliability, efficiency, and accuracy of AI models. “In ten years, half of the world’s jobs will be in prompt engineering,” [claims Robin Li](#), the cofounder and CEO of Chinese tech giant Baidu. However, we expect prompting to be a skill required of many jobs, akin to proficiency in Microsoft Excel, rather than a popular job title in itself. This new wave of disruption is changing everything we thought we knew about computers. We’re used to writing algorithms that return the same result every time—not so for AI, where the responses are non-deterministic. Cost and latency are real factors again, after decades of Moore’s law making us complacent in expecting real-time computation at negligible cost. The biggest hurdle is the tendency of these models to confidently make things up, dubbed *hallucination*, causing us to rethink the way we evaluate the accuracy of our work.

We’ve been working with generative AI since the GPT-3 beta in 2020, and as we saw the models progress, many early prompting tricks and hacks became no longer necessary. Over time a consistent set of principles emerged that were still useful with the newer models, and worked across both text and image generation. We have written this book based on these timeless principles, helping you learn transferable skills that will continue to be useful no matter what happens with AI over the next

five years. The key to working with AI isn’t “figuring out how to hack the prompt by adding one magic word to the end that changes everything else,” as [OpenAI cofounder Sam Altman asserts](#), but what will always matter is the “quality of ideas and the understanding of what you want.” While we don’t know if we’ll call it “prompt engineering” in five years, working effectively with generative AI will only become more important.

## Software Requirements for This Book

All of the code in this book is in Python and was designed to be run in a [Jupyter Notebook](#) or [Google Colab notebook](#). The concepts taught in the book are transferable to JavaScript or any other coding language if preferred, though the primary focus of this book is on prompting techniques rather than traditional coding skills. The code can all be [found on GitHub](#), and we will link to the relevant notebooks throughout. It’s highly recommended that you utilize the [GitHub repository](#) and run the provided examples while reading the book.

For non-notebook examples, you can run the script with the format `python content/chapter_x/script.py` in your terminal, where `x` is the chapter number and `script.py` is the name of the script. In some instances, API keys need to be set as environment variables, and we will make that clear. The packages used update frequently, so install our [`requirements.txt`](#) in a virtual environment before running code examples.

The `requirements.txt` file is generated for Python 3.9. If you want to use a different version of Python, you can generate a new `requirements.txt` from this [`requirements.in`](#) file found within the GitHub repository, by running these commands:

```
'pip install pip-tools'  
'pip-compile requirements.in'
```

For Mac users:

1. Open Terminal: You can find the Terminal application in your Applications folder, under Utilities, or use Spotlight to search for it.
2. Navigate to your project folder: Use the `cd` command to change the directory to your project folder. For example: `cd path/to/your/project`.
3. Create the virtual environment: Use the following command to create a virtual environment named `venv` (you can name it anything): `python3 -m venv venv`.
4. Activate the virtual environment: Before you install packages, you need to activate the virtual environment. Do this with the command `source venv/bin/activate`.

5. Install packages: Now that your virtual environment is active, you can install packages using `pip`. To install packages from the `requirements.txt` file, use `pip install -r requirements.txt`.
6. Deactivate virtual environment: When you're done, you can deactivate the virtual environment by typing `deactivate`.

For Windows users:

1. Open Command Prompt: You can search for `cmd` in the Start menu.
2. Navigate to your project folder: Use the `cd` command to change the directory to your project folder. For example: `cd path\to\your\project`.
3. Create the virtual environment: Use the following command to create a virtual environment named `venv`: `python -m venv venv`.
4. Activate the virtual environment: To activate the virtual environment on Windows, use `.\venv\Scripts\activate`.
5. Install packages: With the virtual environment active, install the required packages: `pip install -r requirements.txt`.
6. Deactivate the virtual environment: To exit the virtual environment, simply type: `deactivate`.

Here are some additional tips on setup:

- Always ensure your Python is up-to-date to avoid compatibility issues.
- Remember to activate your virtual environment whenever you work on the project.
- The `requirements.txt` file should be in the same directory where you create your virtual environment, or you should specify the path to it when using `pip install -r`.

Access to an OpenAI developer account is assumed, as your `OPENAI_API_KEY` must be set as an environment variable in any examples importing the OpenAI library, for which we use version 1.0. Quick-start instructions for setting up your development environment can be found in [OpenAI's documentation](#) on their website.

You must also ensure that *billing is enabled* on your OpenAI account and that a valid payment method is attached to run some of the code within the book. The examples in the book use GPT-4 where not stated, though we do briefly cover Anthropic's competing [Claude 3 model](#), as well as Meta's open source [Llama 3](#) and [Google Gemini](#).

For image generation we use [Midjourney](#), for which you need a Discord account to sign up, though these principles apply equally to DALL-E 3 (available with a ChatGPT Plus subscription or via the API) or Stable Diffusion (available as an [API](#) or it can [run locally](#) on your computer if it has a GPU). The image generation examples in this book use Midjourney v6, Stable Diffusion v1.5 (as many extensions are still only compatible with this version), or [Stable Diffusion XL](#), and we specify the differences when this is important.

We provide examples using open source libraries wherever possible, though we do include commercial vendors where appropriate—for example, [Chapter 5](#) on vector databases demonstrates both FAISS (an open source library) and Pinecone (a paid vendor). The examples demonstrated in the book should be easily modifiable for alternative models and vendors, and the skills taught are transferable. [Chapter 4](#) on advanced text generation is focused on the LLM framework LangChain, and [Chapter 9](#) on advanced image generation is built on AUTOMATIC1111’s open source Stable Diffusion Web UI.

## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

**Constant width**

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**Constant width bold**

Shows commands or other text that should be typed literally by the user.

*Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Throughout the book we reinforce what we call the Five Principles of Prompting, identifying which principle is most applicable to the example at hand. You may want to refer to [Chapter 1](#), which describes the principles in detail.



#### Principle Name

This will explain how the principle is applied to the current example or section of text.

## Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://oreil.ly/prompt-engineering-for-generative-ai>.

If you have a technical question or a problem using the code examples, please send email to [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Prompt Engineering for Generative AI* by James Phoenix and Mike Taylor (O'Reilly). Copyright 2024 Saxifrage, LLC and Just Understanding Data LTD, 978-1-098-15343-4.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at [permissions@oreilly.com](mailto:permissions@oreilly.com).

# O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
800-889-8969 (in the United States or Canada)  
707-827-7019 (international or local)  
707-829-0104 (fax)  
[support@oreilly.com](mailto:support@oreilly.com)  
<https://www.oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/prompt-engineering-generativeAI>.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on LinkedIn: <https://linkedin.com/company/oreilly-media>.

Watch us on YouTube: <https://youtube.com/oreillymedia>.

## Acknowledgments

We'd like to thank the following people for their contribution in conducting a technical review of the book and their patience in correcting a fast-moving target:

- Mayo Oshin, early LangChain contributor and founder at [SeinnAI Analytics](#)
- Ellis Crosby, founder at [Scarlett Panda](#) and AI agency [Incremen.to](#)
- Dave Pawson, O'Reilly author of [XSL-FO](#)

- Mark Phoenix, a senior software engineer
- Aditya Goel, GenAI consultant
- Sanyam Kumar, Associate Director, Data Science, Genmab
- Lakshmanan Sethu, TAM, Gen AI Solutions, Google
- Janit Anjaria, Staff TLM, Aurora Innovation Inc.

We are also grateful to our families for their patience and understanding and would like to reassure them that we still prefer talking to them over ChatGPT.



## CHAPTER 1

# The Five Principles of Prompting

*Prompt engineering* is the process of discovering prompts that reliably yield useful or desired results.

A *prompt* is the input you provide, typically text, when interfacing with an AI model like ChatGPT or Midjourney. The prompt serves as a set of instructions the model uses to predict the desired response: text from *large language models* (LLMs) like [ChatGPT](#), or images from *diffusion models* like [Midjourney](#).

Here is a simple example of a prompt input for a product name generator (inspired by one of [OpenAI's examples](#)), and the resulting output from ChatGPT.

Input:

```
Can I have a list of product names for a pair of shoes that  
can fit any foot size?
```

Output:

```
Certainly! Here's a list of product names for a pair of shoes  
that can fit any foot size:
```

```
UniFit SoleStrides  
FlexiSize All-Fit Shoes  
OmniFit FootFlex  
...  
AnyStep AdaptiWalkers  
FlexiRange UniversalSteps  
PerfectGlide MultiFit Shoes
```

This is already a remarkable response for a naive prompt, which feels like magic because we got here with very little effort. As the state-of-the-art models improve, the likelihood you will get *good enough* results on your first try goes up. For any throwaway interactions with an AI, where you don't plan to do the same task again, the naive approach is all you need.

However, if you planned to put this prompt into production, you'd benefit from investing more work into getting it right. Mistakes cost you money in terms of the fees OpenAI charges based on the length of the prompt and response, as well as the time spent fixing mistakes. If you were building a product name generator with thousands of users, there are some obvious issues you'd want attempt to fix:

#### *Vague direction*

You're not briefing the AI on what style of name you want, or what attributes it should have. Do you want a single word or a concatenation? Can the words be made up, or is it important that they're in real English? Do you want the AI to emulate somebody you admire who is famous for great product names?

#### *Unformatted output*

You're getting back a list of separated names line by line, of unspecified length. When you run this prompt multiple times, you'll see sometimes it comes back with a numbered list, and often it has text at the beginning, which makes it hard to parse programmatically.

#### *Missing examples*

You haven't given the AI any examples of what *good* names look like. It's auto-completing using an average of its training data, i.e., the entire internet (with all its inherent bias), but is that what you want? Ideally you'd feed it examples of successful names, common names in an industry, or even just other names you like.

#### *Limited evaluation*

You have no consistent or scalable way to define which names are good or bad, so you have to manually review each response. If you can institute a rating system or other form of measurement, you can optimize the prompt to get better results and identify how many times it fails.

#### *No task division*

You're asking a lot of a single prompt here: there are lots of factors that go into product naming, and this important task is being naively outsourced to the AI all in one go, with no task specialization or visibility into how it's handling this task for you.

Addressing these problems is the basis for the core principles we use throughout this book. There are many different ways to ask an AI model to do the same task, and even slight changes can make a big difference. LLMs work by continuously predicting the next token (approximately three-fourths of a word), starting from what was in your prompt. Each new token is selected based on its probability of appearing next, with an element of randomness (controlled by the *temperature* parameter). As demonstrated in [Figure 1-1](#), the word *shoes* had a lower probability of coming after the start of the name *AnyFit* (0.88%), where a more predictable response would be *Athletic* (72.35%).

The screenshot shows the OpenAI Playground interface. On the left is a sidebar with icons for user profile, history, save, share, and settings. The main area has a title "Playground" and a text input field containing the prompt: "Can I have a list of product names for a pair of shoes that can fit any foot size?". Below the prompt is a list of generated items, each with a number, name, and probability. A tooltip is shown over item 14, "Shoes = 0.88%", detailing the breakdown of tokens:

Token	Probability (%)
Athletic	72.35%
Boots	8.18%
Tr	5.59%
Athletics	5.42%
Ath	1.63%
O	1.23%
Sand	0.95%
Shoes	0.88%
Sne	0.85%

At the bottom left is a "Submit" button.

*Figure 1-1. How the response breaks down into tokens*

LLMs are trained on essentially the entire text of the internet, and are then further fine-tuned to give helpful responses. Average prompts will return average responses, leading some to be underwhelmed when their results don't live up to the hype. What you put in your prompt changes the probability of every word generated, so it matters a great deal to the results you'll get. These models have seen the best and worst of what humans have produced and are capable of emulating almost anything if you know the right way to ask. OpenAI charges based on the [number of tokens](#)

**used** in the prompt and the response, so prompt engineers need to make these tokens count by optimizing prompts for cost, quality, and reliability.

Here's the same example with the application of several prompt engineering techniques. We ask for names in the style of Steve Jobs, state that we want a comma-separated list, and supply examples of the task done well.

Input:

```
Brainstorm a list of product names for a shoe that fits any
foot size, in the style of Steve Jobs.
```

```
Return the results as a comma-separated list, in this format:
```

```
Product description: A shoe that fits any foot size
```

```
Product names: [list of 3 product names]
```

```
## Examples
```

```
Product description: A refrigerator that dispenses beer
```

```
Product names: iBarFridge, iFridgeBeer, iDrinkBeerFridge
```

```
Product description: A watch that can tell accurate time in
space
```

```
Product names: iNaut, iSpace, iTime
```

```
Product description: A home milkshake maker
```

```
Product names: iShake, iSmoothie, iShake Mini
```

Output:

```
Product description: A shoe that fits any foot size
```

```
Product names: iFitFoot, iPerfectFit, iShoeSize
```

While no prompt is ever perfect, this prompt is optimized to reliably deliver solid product names in the right format. The user of your product name generator can choose somebody other than Steve Jobs to get the types of names they like, they can change the response format if needed, and the output of this prompt can become the input of another. Finally, you could periodically update the examples you use in the prompt based on user feedback, making your system smarter over time.

## Overview of the Five Principles of Prompting

The process for optimizing this prompt follows the *Five Principles of Prompting*, which we will dissect using this example in the remainder of this chapter, and recall throughout the book. They map exactly to the five issues we raised when discussing the naive text prompt. You'll find references back to these principles throughout the rest of the book to help you connect the dots to how they're used in practice. The Five Principles of Prompting are as follows:

### *Give Direction*

Describe the desired style in detail, or reference a relevant persona

### *Specify Format*

Define what rules to follow, and the required structure of the response

### *Provide Examples*

Insert a diverse set of test cases where the task was done correctly

### *Evaluate Quality*

Identify errors and rate responses, testing what drives performance.

### *Divide Labor*

Split tasks into multiple steps, chained together for complex goals

These principles are not short-lived *tips* or *hacks* but are generally accepted conventions that are useful for working with any level of intelligence, biological or artificial. These principles are model-agnostic and should work to improve your prompt no matter which generative text or image model you're using. We first published these principles in July 2022 in the blog post "[Prompt Engineering: From Words to Art and Copy](#)", and they have stood the test of time, including mapping quite closely to OpenAI's own [Prompt Engineering Guide](#), which came a year later. Anyone who works closely with generative AI models is likely to converge on a similar set of strategies for solving common issues, and throughout this book you'll see hundreds of demonstrative examples of how they can be useful for improving your prompts.

We have provided downloadable one-pagers for text and image generation you can use as a checklist when applying these principles. These were created for our popular Udemy course [The Complete Prompt Engineering for AI Bootcamp](#) (70,000+ students), which was based on the same principles but with different material to this book.

- [Text Generation One-Pager](#)
- [Image Generation One-Pager](#)

To show these principles apply equally well to prompting image models, let's use the following example, and explain how to apply each of the Five Principles of Prompting to this specific scenario. Copy and paste the entire input prompt into the Midjourney Bot in Discord, including the link to the image at the beginning, after typing **/imagine** to trigger the prompt box to appear (requires a free [Discord](#) account, and a paid [Midjourney](#) account).

Input:

<https://s.mj.run/TKAsyhNiKmc> stock photo of business meeting of 4 people watching on white MacBook on top of glass-top table, Panasonic, DC-GH5

Figure 1-2 shows the output.



Figure 1-2. Stock photo of business meeting

This prompt takes advantage of Midjourney's ability to take a base image as an example by uploading the image to Discord and then copy and pasting the URL into the prompt (<https://s.mj.run/TKAsyhNiKmc>), for which the royalty-free image from Unsplash is used (Figure 1-3). If you run into an error with the prompt, try uploading the image yourself and reviewing [Midjourney's documentation](#) for any formatting changes.



Figure 1-3. Photo by Mimi Thian on [Unsplash](#)

Let's compare this well-engineered prompt to what you get back from Midjourney if you naively ask for a stock photo in the simplest way possible. [Figure 1-4](#) shows an example of what you get without prompt engineering, an image with a darker, more stylistic take on a stock photo than you'd typically expect.

Input:

```
people in a business meeting
```

[Figure 1-4](#) shows the output.

Although less prominent an issue in v5 of Midjourney onwards, community feedback mechanisms (when users select an image to resize to a higher resolution, that choice may be used to train the model) have reportedly biased the model toward a *fantasy* aesthetic, which is less suitable for the stock photo use case. The early adopters of Midjourney came from the digital art world and naturally gravitated toward fantasy and sci-fi styles, which can be reflected in the results from the model even when this aesthetic is not suitable.



Figure 1-4. People in a business meeting

Throughout this book the examples used will be compatible with ChatGPT Plus (GPT-4) as the text model and Midjourney v6 or Stable Diffusion XL as the image model, though we will specify if it's important. These foundational models are the current state of the art and are good at a diverse range of tasks. The principles are intended to be future-proof as much as is possible, so if you're reading this book when GPT-5, Midjourney v7, or Stable Diffusion XXL is out, or if you're using another vendor like Google, everything you learn here should still prove useful.

## 1. Give Direction

One of the issues with the naive text prompt discussed earlier was that it wasn't briefing the AI on what *types* of product names you wanted. To some extent, naming a product is a subjective endeavor, and without giving the AI an idea of what names you like, it has a low probability of guessing right.

By the way, a human would also struggle to complete this task without a good *brief*, which is why creative and branding agencies require a detailed briefing on any task from their clients.



Although it's not a perfect mapping, it can be helpful to imagine what context a human might need for this task and try including it in the prompt.

In the example prompt we gave direction through the use of *role-playing*, in that case emulating the style of Steve Jobs, who was famous for iconically naming products. If you change this aspect of the prompt to someone else who is famous in the training data (as well as matching the examples to the right style), you'll get dramatically different results.

Input:

```
Brainstorm a list of product names for a shoe that fits any
foot size, in the style of Elon Musk.
```

Return the results as a comma-separated list, in this format:

Product description: A shoe that fits any foot size

Product names: [list of 3 product names]

## Examples

Product description: A refrigerator that dispenses beer

Product names: Teslacool, FridgeX, Neuradrink

Product description: A watch that can tell accurate time in
space

Product names: WatchX, iSpace, The Watch Company

Product description: A home milkshake maker

Product names: ShakeX, The Milkshake Company, HyperShake

Output:

ShoeX, MarsFit, The Shoe Company

There are also some rules or best practices you would do well to follow, which could be included in the prompt as context to guide the AI toward a name that works. This technique is sometimes referred to as *prewarming* or *internal retrieval*, and it is simple but effective (Liu et al., 2021). Starting the conversation asking for best practice advice, then asking it to follow its own advice, can help a lot. In effect, you are using it to generate its own direction.

Input:

Please give me 5 tips for naming products based on expert industry advice.

Output:

Certainly! Naming a product is crucial as it often gives the first impression to potential customers. Here are five expert tips to guide you in naming products:

1. **\*\*Simplicity & Memorability\*\***: Opt for names that are short, easy to spell, and memorable. Long, complicated names can be difficult for customers to recall. Remember, the easier the name is to remember, the more likely it is that consumers will think of your product when making a purchase decision.

...

5. **\*\*Positive Connotations\*\***: Names that evoke positive emotions or imagery can create a favorable perception of your product. Avoid names that have negative or controversial associations in any of the primary languages or cultures where you intend to do business. It's often worthwhile to conduct linguistic checks and market testing to ensure there aren't unintended meanings.

Lastly, once you've shortlisted a few names, it's a good idea to run them by diverse focus groups or conduct surveys to gauge reactions. Also, ensure that the name you choose is available for trademarking and doesn't infringe on existing trademarks.

Then within the same chat window, where the model has the context of the past advice it gave, you ask your initial prompt for the task you wanted to complete.

Input:

Using this advice, can I have a list of product names for a pair of shoes that can fit any foot size?

Output:

Certainly! Based on the criteria given:

1. **\*\*Simplicity & Memorability\*\***
2. **\*\*Relevance & Descriptiveness\*\***
3. **\*\*Differentiation & Uniqueness\*\***
4. **\*\*Adaptable & Scalable\*\***
5. **\*\*Positive Connotations\*\***

Here's a list of potential product names for shoes that can fit any foot size:

1. **\*\*FitAll Walkers\*\***: A simple and memorable name that instantly suggests a universal fit.

...

10. **\*\*TrueFit Universals\*\***: Emphasizing genuine comfort and fit for all.

Before finalizing any name, remember to check for trademarks to ensure there's no conflict with existing products or brands.

Another fruitful strategy in our experience is to take the best advice out there for the task you want to accomplish and insert that context into the prompt. For example, you could take [Brandwatch's 5 Golden Rules for naming a product](#) or another trusted external resource you find, and insert that as context into the prompt. This will increase the length of the prompt significantly, which costs more money (when using the API as a developer), but may be worth the trade-off if the quality of the response improves.

Input:

Can I have a list of product names for a pair of shoes that can fit any foot size? Please follow the five rules below:

1. It should be readable and writable.

If your product name is hard to pronounce, people won't talk about it, and if they can't write it down (and spell it correctly!) when they hear it, how do you expect them to Google it?

Keep it simple and don't go with any wacky spellings just for the sake of it.

...

5. It should evoke an emotion, feeling, or idea.

Your product name should tie back into what your product is, what the feeling you want people to have when experiencing your product is, and/or what idea are you trying to get across.

It should be emotive and inspiring.

Output:

1. FlexFit Footwear

2. OmniSize Sneakers

3. AdaptStride Shoes

...

18. OmniComfort Kicks

19. FlexSize Footwear

20. Boundless Soles

There are other myriad ways of providing direction. In the image generation example, direction was given by specifying that the business meeting is taking place around a glass-top table. If you change only that detail, you can get a completely different image, as detailed in [Figure 1-5](#).

Input:

<https://s.mj.run/TKAsyhNiKmc> stock photo of business meeting of four people gathered around a campfire outdoors in the woods, Panasonic, DC-GH5

[Figure 1-5](#) shows the output.



*Figure 1-5. Stock photo of business meeting in the woods*

Role-playing is also important for image generation, and one of the quite powerful ways you can give Midjourney direction is to supply the name of an artist or art style to emulate. One artist that features heavily in the AI art world is Van Gogh, known for his bold, dramatic brush strokes and vivid use of colors. Watch what happens when you include his name in the prompt, as shown in [Figure 1-6](#).

Input:

people in a business meeting, by Van Gogh

Figure 1-6 shows the output.



Figure 1-6. People in a business meeting, by Van Gogh

To get that last prompt to work, you need to strip back a lot of the other direction. For example, losing the base image and the words *stock photo* as well as the camera *Panasonic, DC-GH5* helps bring in Van Gogh's style. The problem you may run into is that often with too much direction, the model can quickly get to a conflicting combination that it can't resolve. If your prompt is overly specific, there might not be enough samples in the training data to generate an image that's consistent with all of your criteria. In cases like these, you should choose which element is more important (in this case, Van Gogh) and defer to that.

Direction is one of the most commonly used and broadest principles. It can take the form of simply using the right descriptive words to clarify your intent, or channeling the personas of relevant business celebrities. While too much direction can narrow the creativity of the model, too little direction is the more common problem.

## 2. Specify Format

AI models are universal translators. Not only does that mean translating from French to English, or Urdu to Klingon, but also between data structures like JSON to YAML, or natural language to Python code. These models are capable of returning a response in almost any format, so an important part of prompt engineering is finding ways to specify what format you want the response to be in.

Every now and again you'll find that the same prompt will return a different format, for example, a numbered list instead of comma separated. This isn't a big deal most of the time, because most prompts are one-offs and typed into ChatGPT or Midjourney. However, when you're incorporating AI tools into production software, occasional flips in format can cause all kinds of errors.

Just like when working with a human, you can avoid wasted effort by specifying up front the format you expect the response to be in. For text generation models, it can often be helpful to output JSON instead of a simple ordered list because that's the universal format for API responses, which can make it simpler to parse and spot errors, as well as to use to render the front-end HTML of an application. YAML is also another popular choice because it enforces a parseable structure while still being simple and human-readable.

In the original prompt you gave direction through both the examples provided, and the colon at the end of the prompt indicated it should complete the list inline. To swap the format to JSON, you need to update both and leave the JSON uncompleted, so GPT-4 knows to complete it.

Input:

```
Return a comma-separated list of product names in JSON for
"A pair of shoes that can fit any foot size.".
Return only JSON.
```

Examples:

```
[{
    "Product description": "A home milkshake maker.",
    "Product names": ["HomeShaker", "Fit Shaker",
                     "QuickShake", "Shake Maker"]
},
{
    "Product description": "A watch that can tell
                          accurate time in space.",
    "Product names": ["AstroTime", "SpaceGuard",
                     "Orbit-Accurate", "EliptoTime"]}
]
```

Output:

```
[  
 {  
   "Product description": "A pair of shoes that can \  
 fit any foot size.",  
   "Product names": ["FlexFit Footwear", "OneSize Step",  
   "Adapt-a-Shoe", "Universal Walker"]  
 }  
 ]
```

The output we get back is the completed JSON containing the product names. This can then be parsed and used programmatically, in an application or local script. It's also easy from this point to check if there's an error in the formatting using a JSON parser like Python's standard *json* library, because broken JSON will result in a parsing error, which can act as a trigger to retry the prompt or investigate before continuing. If you're still not getting the right format back, it can help to specify at the beginning or end of the prompt, or in the system message if using a chat model: You are a helpful assistant that only responds in JSON, or specify JSON output in the model parameters where available (this is called *grammars* with *Llama models*.



To get up to speed on JSON if you're unfamiliar, W3Schools has a [good introduction](#).

For image generation models, format is very important, because the opportunities for modifying an image are near endless. They range from obvious formats like stock photo, illustration, and oil painting, to more unusual formats like dashcam footage, ice sculpture, or in Minecraft (see [Figure 1-7](#)).

Input:

```
business meeting of four people watching on MacBook on top of  
table, in Minecraft
```

[Figure 1-7](#) shows the output.



Figure 1-7. Business meeting in Minecraft

When setting a format, it is often necessary to remove other aspects of the prompt that might clash with the specified format. For example, if you supply a base image of a stock photo, the result is some combination of stock photo and the format you wanted. To some degree, image generation models can generalize to new scenarios and combinations they haven't seen before in their training set, but in our experience, the more layers of unrelated elements, the more likely you are to get an unsuitable image.

There is often some overlap between the first and second principles, Give Direction and Specify Format. The latter is about defining what type of output you want, for example JSON format, or the format of a stock photo. The former is about the style of response you want, independent from the format, for example product names in the style of Steve Jobs, or an image of a business meeting in the style of Van Gogh. When there are clashes between style and format, it's often best to resolve them by dropping whichever element is less important to your final result.

### 3. Provide Examples

The original prompt didn't give the AI any examples of what you think *good* names look like. Therefore, the response is approximate to an average of the internet, and you can do better than that. Researchers would call a prompt with no examples *zero-shot*, and it's always a pleasant surprise when AI can even do a task zero shot: it's a sign of a powerful model. If you're providing zero examples, you're asking for a lot without giving much in return. Even providing one example (*one-shot*) helps considerably, and it's the norm among researchers to test how models perform with multiple examples (*few-shot*). One such piece of research is the famous GPT-3 paper "[Language Models are Few-Shot Learners](#)", the results of which are illustrated in [Figure 1-8](#), showing adding one example along with a prompt can improve accuracy in some tasks from 10% to near 50%!

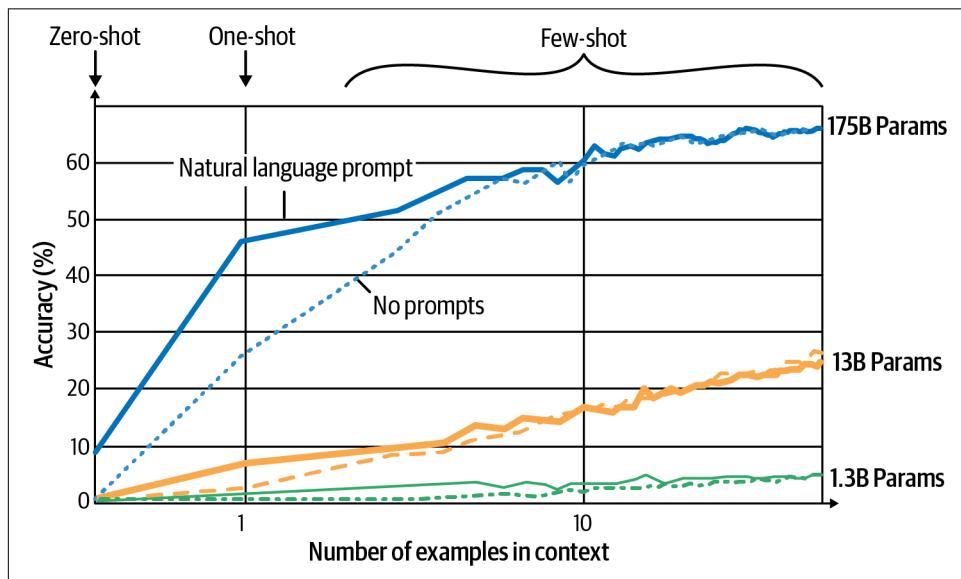


Figure 1-8. Number of examples in context

When briefing a colleague or training a junior employee on a new task, it's only natural that you'd include examples of times that task had previously been done well. Working with AI is the same, and the strength of a prompt often comes down to the examples used. Providing examples can sometimes be easier than trying to explain exactly what it is about those examples you like, so this technique is most effective when you are not a domain expert in the subject area of the task you are attempting to complete. The amount of text you can fit in a prompt is limited (at the time of writing around 6,000 characters on Midjourney and approximately 32,000 characters for the free version of ChatGPT), so a lot of the work of prompt engineering involves selecting and inserting diverse and instructive examples.

There's a trade-off between reliability and creativity: go past three to five examples and your results will become more reliable, while sacrificing creativity. The more examples you provide, and the lesser the diversity between them, the more constrained the response will be to match your examples. If you change all of the examples to animal names in the previous prompt, you'll have a strong effect on the response, which will reliably return only names including animals.

Input:

```
Brainstorm a list of product names for a shoe that fits any foot size.
```

```
Return the results as a comma-separated list, in this format:
```

```
Product description: A shoe that fits any foot size
```

```
Product names: [list of 3 product names]
```

```
## Examples:
```

```
Product description: A home milkshake maker.
```

```
Product names: Fast Panda, Healthy Bear, Compact Koala
```

```
Product description: A watch that can tell accurate time in space.
```

```
Product names: AstroLamb, Space Bear, Eagle Orbit
```

```
Product description: A refrigerator that dispenses beer
```

```
Product names: BearFridge, Cool Cat, PenguinBox
```

Output:

```
Product description: A shoe that fits any foot size
```

```
Product names: FlexiFox, ChameleonStep, PandaPaws
```

Of course this runs the risk of missing out on returning a much better name that doesn't fit the limited space left for the AI to play in. Lack of diversity and variation in examples is also a problem in handling edge cases, or uncommon scenarios. Including one to three examples is easy and almost always has a positive effect, but above that number it becomes essential to experiment with the number of examples you include, as well as the similarity between them. There is some evidence ([Hsieh et al., 2023](#)) that direction works better than providing examples, and it typically isn't straightforward to collect good examples, so it's usually prudent to attempt the principle of Give Direction first.

In the image generation space, providing examples usually comes in the form of providing a base image in the prompt, called *img2img* in the open source [Stable Diffusion](#) community. Depending on the image generation model being used, these images can be used as a starting point for the model to generate from, which greatly affects the results. You can keep everything about the prompt the same but swap out the provided base image for a radically different effect, as in [Figure 1-9](#).

Input:

stock photo of business meeting of 4 people watching on white MacBook on top of glass-top table, Panasonic, DC-GH5

Figure 1-9 shows the output.

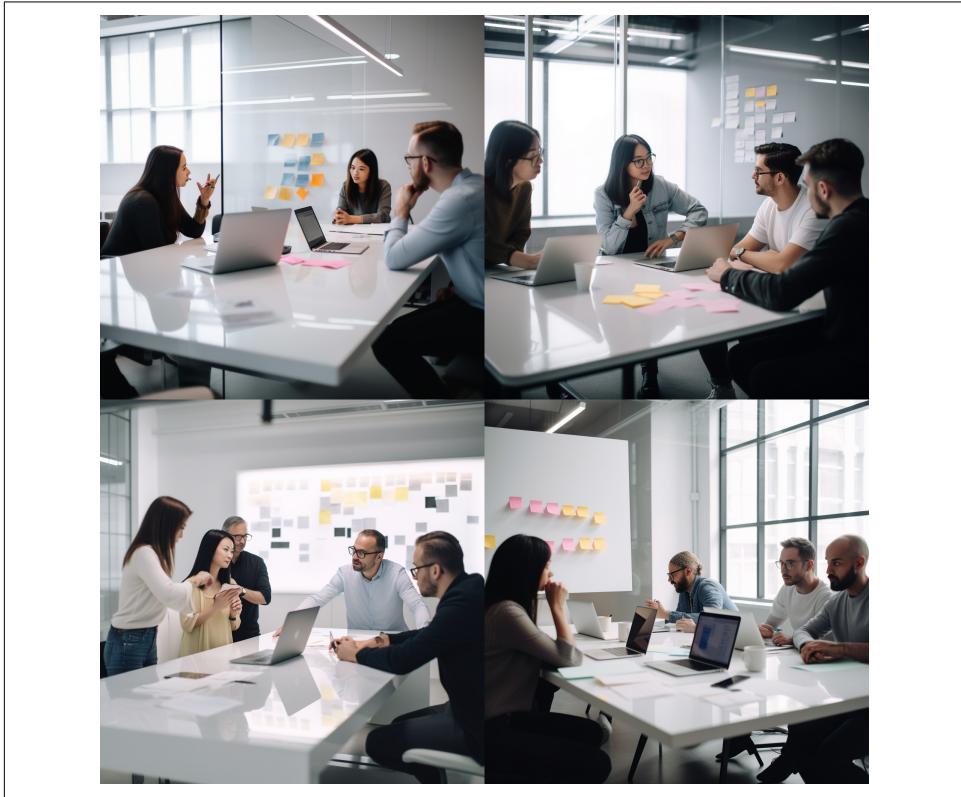


Figure 1-9. Stock photo of business meeting of four people

In this case, by substituting for the image shown in Figure 1-10, also from Unsplash, you can see how the model was pulled in a different direction and incorporates whiteboards and sticky notes now.



These examples demonstrate the capabilities of image generation models, but we would exercise caution when uploading base images for use in prompts. Check the licensing of the image you plan to upload and use in your prompt as the base image, and avoid using clearly copyrighted images. Doing so can land you in legal trouble and is against the terms of service for all the major image generation model providers.



Figure 1-10. Photo by Jason Goodman on [Unsplash](#)

## 4. Evaluate Quality

As of yet, there has been no feedback loop to judge the quality of your responses, other than the basic trial and error of running the prompt and seeing the results, referred to as *blind prompting*. This is fine when your prompts are used temporarily for a single task and rarely revisited. However, when you're reusing the same prompt multiple times or building a production application that relies on a prompt, you need to be more rigorous with measuring results.

There are a number of ways performance can be evaluated, and it depends largely on what tasks you're hoping to accomplish. When a new AI model is released, the focus tends to be on how well the model did on *evals* (evaluations), a standardized set of questions with predefined answers or grading criteria that are used to test performance across models. Different models perform differently across different types of tasks, and there is no guarantee a prompt that worked previously will translate well to a new model. OpenAI has [made its evals framework](#) for benchmarking performance of LLMs open source and encourages others to contribute additional eval templates.

In addition to the standard academic evals, there are also more headline-worthy tests like [GPT-4 passing the bar exam](#). Evaluation is difficult for more subjective tasks, and can be time-consuming or prohibitively costly for smaller teams. In some instances researchers have turned to using more advanced models like GPT-4 to evaluate responses from less sophisticated models, as was done with [the release of Vicuna-13B](#), a fine-tuned model based on Meta's Llama open source model (see [Figure 1-11](#)).

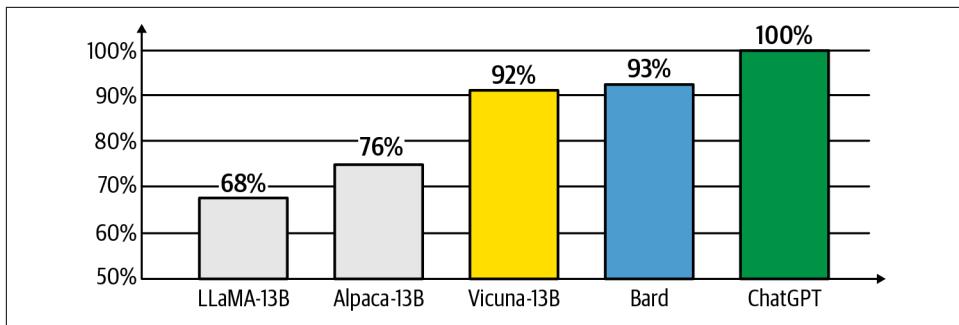


Figure 1-11. Vicuna GPT-4 Evals

More rigorous evaluation techniques are necessary when writing scientific papers or grading a new foundation model release, but often you will only need to go just one step above basic trial and error. You may find that a simple thumbs-up/thumbs-down rating system implemented in a Jupyter Notebook can be enough to add some rigor to prompt optimization, without adding too much overhead. One common test is to see whether providing examples is worth the additional cost in terms of prompt length, or whether you can get away with providing no examples in the prompt. The first step is getting responses for multiple runs of each prompt and storing them in a spreadsheet, which we will do after setting up our environment.

You can install the OpenAI Python package with `pip install openai`. If you're running into compatibility issues with this package, create a virtual environment and install our [requirements.txt](#) (instructions in the preface).

To utilize the API, you'll need to [create an OpenAI account](#) and then [navigate here for your API key](#).



Hardcoding API keys in scripts is not recommended due to security reasons. Instead, utilize environment variables or configuration files to manage your keys.

Once you have an API key, it's crucial to assign it as an environment variable by executing the following command, replacing `api_key` with your actual API key value:

```
export OPENAI_API_KEY="api_key"
```

Or on Windows:

```
set OPENAI_API_KEY=api_key
```

Alternatively, if you'd prefer not to preset an API key, then you can manually set the key while initializing the model, or load it from an `.env` file using `python-dotenv`. First, install the library with `pip install python-dotenv`, and then load the environment variables with the following code at the top of your script or notebook:

```
from dotenv import load_dotenv

load_dotenv() # take environment variables from .env.
```

The first step is getting responses for multiple runs of each prompt and storing them in a spreadsheet.

Input:

```
# Define two variants of the prompt to test zero-shot
# vs few-shot
prompt_A = """Product description: A pair of shoes that can
fit any foot size.
Seed words: adaptable, fit, omni-fit.
Product names:"""

prompt_B = """Product description: A home milkshake maker.
Seed words: fast, healthy, compact.
Product names: HomeShaker, Fit Shaker, QuickShake, Shake
Maker

Product description: A watch that can tell accurate time in
space.
Seed words: astronaut, space-hardened, elliptical orbit
Product names: AstroTime, SpaceGuard, Orbit-Accurate,
EliptoTime.

Product description: A pair of shoes that can fit any foot
size.
Seed words: adaptable, fit, omni-fit.
Product names:"""

test_prompts = [prompt_A, prompt_B]

import pandas as pd
from openai import OpenAI
import os

# Set your OpenAI key as an environment variable
# https://platform.openai.com/api-keys
client = OpenAI(
    api_key=os.environ['OPENAI_API_KEY'], # Default
)

def get_response(prompt):
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
```

```

messages=[
    {
        "role": "system",
        "content": "You are a helpful assistant."
    },
    {
        "role": "user",
        "content": prompt
    }
]
)
return response.choices[0].message.content

# Iterate through the prompts and get responses
responses = []
num_tests = 5

for idx, prompt in enumerate(test_prompts):
    # prompt number as a letter
    var_name = chr(ord('A') + idx)

    for i in range(num_tests):
        # Get a response from the model
        response = get_response(prompt)

        data = {
            "variant": var_name,
            "prompt": prompt,
            "response": response
        }
        responses.append(data)

# Convert responses into a dataframe
df = pd.DataFrame(responses)

# Save the dataframe as a CSV file
df.to_csv("responses.csv", index=False)

print(df)

```

Output:

	variant	prompt
0	A	Product description: A pair of shoes that can ...
1	A	Product description: A pair of shoes that can ...
2	A	Product description: A pair of shoes that can ...
3	A	Product description: A pair of shoes that can ...
4	A	Product description: A pair of shoes that can ...
5	B	Product description: A home milkshake maker.\n...
6	B	Product description: A home milkshake maker.\n...
7	B	Product description: A home milkshake maker.\n...
8	B	Product description: A home milkshake maker.\n...

```

9      B Product description: A home milkshake maker.\n...
                                         response
0  1. Adapt-a-Fit Shoes \n2. Omni-Fit Footwear \n...
1  1. OmniFit Shoes\n2. Adapt-a-Sneaks \n3. OneFi...
2  1. Adapt-a-fit\n2. Flexi-fit shoes\n3. Omni-fe...
3  1. Adapt-A-Sole\n2. FitFlex\n3. Omni-FitX\n4. ...
4  1. Omni-Fit Shoes\n2. Adapt-a-Fit Shoes\n3. An...
5  Adapt-a-Fit, Perfect Fit Shoes, OmniShoe, OneS...
6      FitAll, OmniFit Shoes, SizeLess, AdaptaShoes
7      AdaptaFit, OmniShoe, PerfectFit, AllSizeFit.
8  FitMaster, AdaptoShoe, OmniFit, AnySize Footwe...
9      Adapta-Shoe, PerfectFit, OmniSize, FitForm

```

Here we're using the OpenAI API to generate model responses to a set of prompts and storing the results in a dataframe, which is saved to a CSV file. Here's how it works:

1. Two prompt variants are defined, and each variant consists of a product description, seed words, and potential product names, but `prompt_B` provides two examples.
2. Import statements are called for the Pandas library, OpenAI library, and os library.
3. The `get_response` function takes a prompt as input and returns a response from the gpt-3.5-turbo model. The prompt is passed as a user message to the model, along with a system message to set the model's behavior.
4. Two prompt variants are stored in the `test_prompts` list.
5. An empty list `responses` is created to store the generated responses, and the variable `num_tests` is set to 5.
6. A nested loop is used to generate responses. The outer loop iterates over each prompt, and the inner loop generates `num_tests` (five in this case) number of responses per prompt.
  - a. The `enumerate` function is used to get the index and value of each prompt in `test_prompts`. This index is then converted to a corresponding uppercase letter (e.g., 0 becomes A, 1 becomes B) to be used as a variant name.
  - b. For each iteration, the `get_response` function is called with the current prompt to generate a response from the model.
  - c. A dictionary is created with the variant name, the prompt, and the model's response, and this dictionary is appended to the `responses` list.
7. Once all responses have been generated, the `responses` list (which is now a list of dictionaries) is converted into a Pandas DataFrame.

- This dataframe is then saved to a CSV file with the Pandas built-in `to_csv` function, making the file `responses.csv` with `index=False` so as to not write row indices.
- Finally, the dataframe is printed to the console.

Having these responses in a spreadsheet is already useful, because you can see right away even in the printed response that `prompt_A` (zero-shot) in the first five rows is giving us a numbered list, whereas `prompt_B` (few-shot) in the last five rows tends to output the desired format of a comma-separated inline list. The next step is to give a rating on each of the responses, which is best done blind and randomized to avoid favoring one prompt over another.

Input:

```
import ipywidgets as widgets
from IPython.display import display
import pandas as pd

# load the responses.csv file
df = pd.read_csv("responses.csv")

# Shuffle the dataframe
df = df.sample(frac=1).reset_index(drop=True)

# df is your dataframe and 'response' is the column with the
# text you want to test
response_index = 0
# add a new column to store feedback
df['feedback'] = pd.Series(dtype='str')

def on_button_clicked(b):
    global response_index
    # convert thumbs up / down to 1 / 0
    user_feedback = 1 if b.description == "\ud83d\udc4d" else 0

    # update the feedback column
    df.at[response_index, 'feedback'] = user_feedback

    response_index += 1
    if response_index < len(df):
        update_response()
    else:
        # save the feedback to a CSV file
        df.to_csv("results.csv", index=False)

        print("A/B testing completed. Here's the results:")
        # Calculate score and num rows for each variant
        summary_df = df.groupby('variant').agg(
            count=('feedback', 'count'),
            score=('feedback', 'mean')).reset_index()
```

```

print(summary_df)

def update_response():
    new_response = df.iloc[response_index]['response']
    if pd.notna(new_response):
        new_response = "<p>" + new_response + "</p>"
    else:
        new_response = "<p>No response</p>"
    response.value = new_response
    count_label.value = f"Response: {response_index + 1}"
    count_label.value += f"/{len(df)}"

response = widgets.HTML()
count_label = widgets.Label()

update_response()

thumbs_up_button = widgets.Button(description='\U0001F44D')
thumbs_up_button.on_click(on_button_clicked)

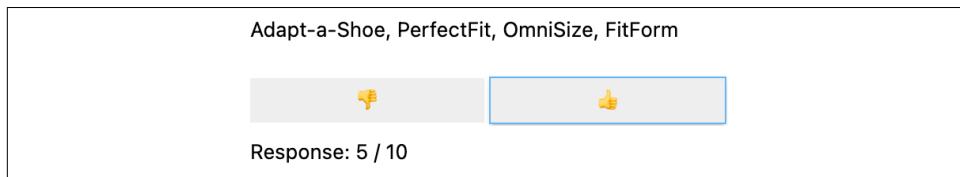
thumbs_down_button = widgets.Button(
    description='\U0001F44E')
thumbs_down_button.on_click(on_button_clicked)

button_box = widgets.HBox([thumbs_down_button,
thumbs_up_button])

display(response, button_box, count_label)

```

The output is shown in Figure 1-12:



*Figure 1-12. Thumbs-up/thumbs-down rating system*

If you run this in a Jupyter Notebook, a widget displays each AI response, with a thumbs-up or thumbs-down button (see Figure 1-12). This provides a simple interface for quickly labeling responses, with minimal overhead. If you wish to do this outside of a Jupyter Notebook, you could change the thumbs-up and thumbs-down emojis for Y and N, and implement a loop using the built-in `input()` function, as a text-only replacement for iPyWidgets.

Once you've finished labeling the responses, you get the output, which shows you how each prompt performs.

Output:

```
A/B testing completed. Here's the results:  
   variant  count   score  
0        A      5    0.2  
1        B      5    0.6
```

The dataframe was shuffled at random, and each response was labeled blind (without seeing the prompt), so you get an accurate picture of how often each prompt performed. Here is the step-by-step explanation:

1. Three modules are imported: `ipywidgets`, `IPython.display`, and `pandas`. `ipywidgets` contains interactive HTML widgets for Jupyter Notebooks and the IPython kernel. `IPython.display` provides classes for displaying various types of output like images, sound, displaying HTML, etc. Pandas is a powerful data manipulation library.
2. The `pandas` library is used to read in the CSV file `responses.csv`, which contains the responses you want to test. This creates a Pandas DataFrame called `df`.
3. `df` is shuffled using the `sample()` function with `frac=1`, which means it uses all the rows. The `reset_index(drop=True)` is used to reset the indices to the standard 0, 1, 2, ..., n index.
4. The script defines `response_index` as 0. This is used to track which response from the dataframe the user is currently viewing.
5. A new column `feedback` is added to the dataframe `df` with the data type as `str` or string.
6. Next, the script defines a function `on_button_clicked(b)`, which will execute whenever one of the two buttons in the interface is clicked.
  - a. The function first checks the `description` of the button clicked was the thumbs-up button (\U0001F44D; ) , and sets `user_feedback` as 1, or if it was the thumbs-down button (\U0001F44E ) , it sets `user_feedback` as 0.
  - b. Then it updates the `feedback` column of the dataframe at the current `response_index` with `user_feedback`.
  - c. After that, it increments `response_index` to move to the next response.
  - d. If `response_index` is still less than the total number of responses (i.e., the length of the dataframe), it calls the function `update_response()`.
  - e. If there are no more responses, it saves the dataframe to a new CSV file `results.csv`, then prints a message, and also prints a summary of the results by variant, showing the count of feedback received and the average score (mean) for each variant.

7. The function `update_response()` fetches the next response from the dataframe, wraps it in paragraph HTML tags (if it's not null), updates the `response` widget to display the new response, and updates the `count_label` widget to reflect the current response number and total number of responses.
8. Two widgets, `response` (an HTML widget) and `count_label` (a Label widget), are instantiated. The `update_response()` function is then called to initialize these widgets with the first response and the appropriate label.
9. Two more widgets, `thumbs_up_button` and `thumbs_down_button` (both Button widgets), are created with thumbs-up and thumbs-down emoji as their descriptions, respectively. Both buttons are configured to call the `on_button_clicked()` function when clicked.
10. The two buttons are grouped into a horizontal box (`button_box`) using the `HBox` function.
11. Finally, the `response`, `button_box`, and `count_label` widgets are displayed to the user using the `display()` function from the `IPython.display` module.

A simple rating system such as this one can be useful in judging prompt quality and encountering edge cases. Usually in less than 10 test runs of a prompt you uncover a deviation, which you otherwise wouldn't have caught until you started using it in production. The downside is that it can get tedious rating lots of responses manually, and your ratings might not represent the preferences of your intended audience. However, even small numbers of tests can reveal large differences between two prompting strategies and reveal nonobvious issues before reaching production.

Iterating on and testing prompts can lead to radical decreases in the length of the prompt and therefore the cost and latency of your system. If you can find another prompt that performs equally as well (or better) but uses a shorter prompt, you can afford to scale up your operation considerably. Often you'll find in this process that many elements of a complex prompt are completely superfluous, or even counterproductive.

The *thumbs-up* or other manually labeled indicators of quality don't have to be the only judging criteria. Human evaluation is generally considered to be the most accurate form of feedback. However, it can be tedious and costly to rate many samples manually. In many cases, as in math or classification use cases, it may be possible to establish *ground truth* (reference answers to test cases) to programmatically rate the results, allowing you to scale up considerably your testing and monitoring efforts. The following is not an exhaustive list because there are many motivations for evaluating your prompt programmatically:

### *Cost*

Prompts that use a lot of tokens, or work only with more expensive models, might be impractical for production use.

### *Latency*

Equally the more tokens there are, or the larger the model required, the longer it takes to complete a task, which can harm user experience.

### *Calls*

Many AI systems require multiple calls in a loop to complete a task, which can seriously slow down the process.

### *Performance*

Implement some form of external feedback system, for example a physics engine or other model for predicting real-world results.

### *Classification*

Determine how often a prompt correctly labels given text, using another AI model or rules-based labeling.

### *Reasoning*

Work out which instances the AI fails to apply logical reasoning or gets the math wrong versus reference cases.

### *Hallucinations*

See how frequently you encounter hallucinations, as measured by invention of new terms not included in the prompt's context.

### *Safety*

Flag any scenarios where the system might return unsafe or undesirable results using a safety filter or detection system.

### *Refusals*

Find out how often the system incorrectly refuses to fulfill a reasonable user request by flagging known refusal language.

### *Adversarial*

Make the prompt robust against known **prompt injection** attacks that can get the model to run undesirable prompts instead of what you programmed.

### *Similarity*

Use shared words and phrases (**BLEU** or **ROGUE**) or vector distance (explained in **Chapter 5**) to measure similarity between generated and reference text.

Once you start rating which examples were good, you can more easily update the examples used in your prompt as a way to continuously make your system smarter over time. The data from this feedback can also feed into examples for fine-tuning, which starts to beat prompt engineering once you can **supply a few thousand examples**, as shown in Figure 1-13.

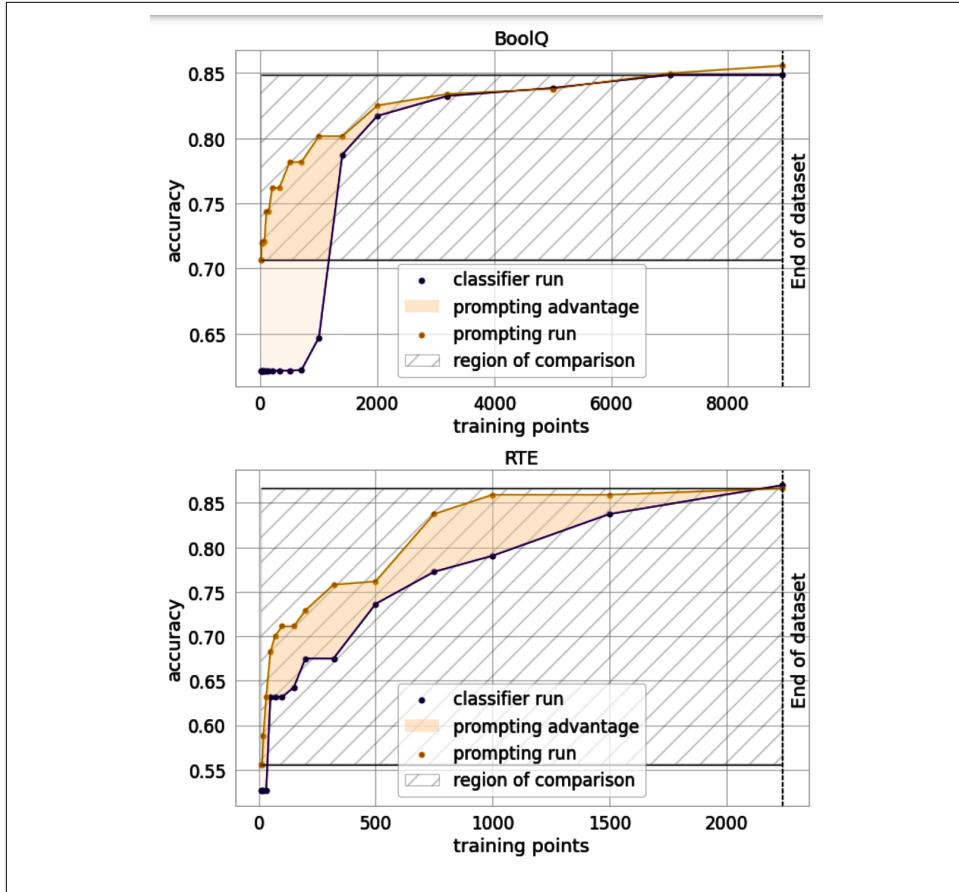


Figure 1-13. How many data points is a prompt worth?

Graduating from thumbs-up or thumbs-down, you can implement a 3-, 5-, or 10-point rating system to get more fine-grained feedback on the quality of your prompts. It's also possible to determine aggregate relative performance through comparing responses side by side, rather than looking at responses one at a time. From this you can construct a fair across-model comparison using an *Elo rating*, as is popular in chess and used in the [Chatbot Arena](#) by [lmsys.org](#).

For image generation, evaluation usually takes the form of *permutation prompting*, where you input multiple directions or formats and generate an image for each

combination. Images can than be scanned or later arranged in a grid to show the effect that different elements of the prompt can have on the final image.

Input:

```
{stock photo, oil painting, illustration} of business  
meeting of {four, eight} people watching on white MacBook on  
top of glass-top table
```

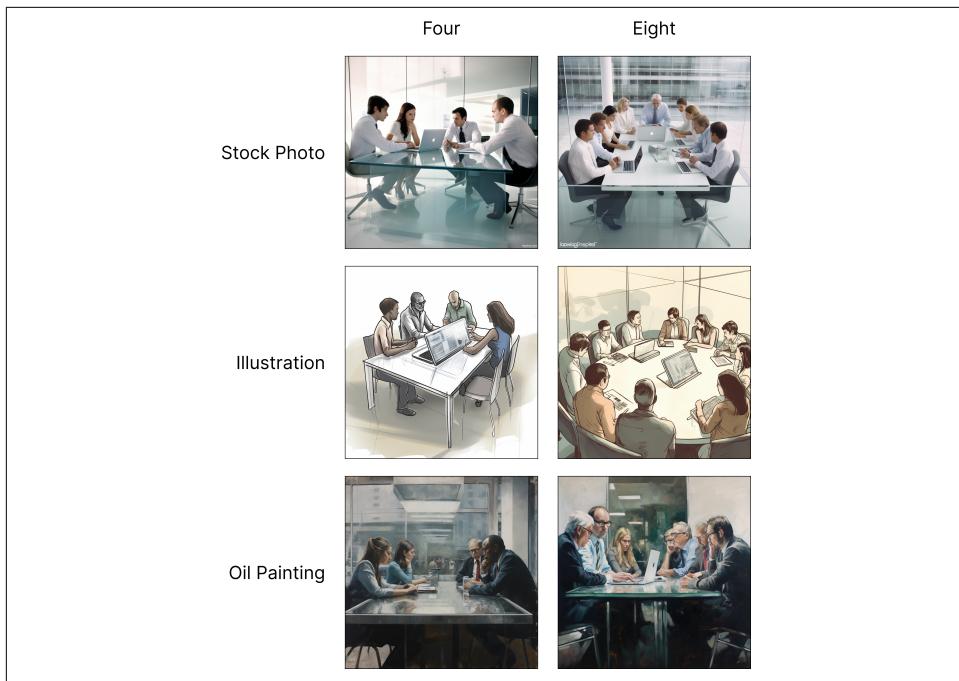
In Midjourney this would be compiled into six different prompts, one for every combination of the three formats (stock photo, oil painting, illustration) and two numbers of people (four, eight).

Input:

1. stock photo of business meeting of four people watching on white MacBook on top of glass-top table
2. stock photo of business meeting of eight people watching on white MacBook on top of glass-top table
3. oil painting of business meeting of four people watching on white MacBook on top of glass-top table
4. oil painting of business meeting of eight people watching on white MacBook on top of glass-top table
5. illustration of business meeting of four people watching on white MacBook on top of glass-top table
6. illustration of business meeting of eight people watching on white MacBook on top of glass-top table

Each prompt generates its own four images as usual, which makes the output a little harder to see. We have selected one from each prompt to upscale and then put them together in a grid, shown as [Figure 1-14](#). You'll notice that the model doesn't always get the correct number of people (generative AI models are surprisingly bad at math), but it has correctly inferred the general intention by adding more people to the photos on the right than the left.

[Figure 1-14](#) shows the output.



*Figure 1-14. Prompt permutations grid*

With models that have APIs like Stable Diffusion, you can more easily manipulate the photos and display them in a grid format for easy scanning. You can also manipulate the random seed of the image to fix a style in place for maximum reproducibility. With image classifiers it may also be possible to programmatically rate images based on their safe content, or if they contain certain elements associated with success or failure.

## 5. Divide Labor

As you build out your prompt, you start to get to the point where you're asking a lot in a single call to the AI. When prompts get longer and more convoluted, you may find the responses get less deterministic, and hallucinations or anomalies increase. Even if you manage to arrive at a reliable prompt for your task, that task is likely just one of a number of interrelated tasks you need to do your job. It's natural to start exploring how many other of these tasks could be done by AI and how you might string them together.

One of the core principles of engineering is to use task decomposition to break problems down into their component parts, so you can more easily solve each individual problem and then reaggregate the results. Breaking your AI work into multiple calls that are chained together can help you accomplish more complex tasks, as well as provide more visibility into what part of the chain is failing.

There are lots of factors that go into product naming, and an important task is naively outsourced to the AI with no visibility into how it's weighing the importance of these factors (if at all). The way our current system works, we're getting a list of names, but all names are displayed with equal importance with no further context for helping us decide. Fortunately AI tools are capable of self-evaluation; if we add a second step to our task, we can automatically check for undesirable outputs.

Input:

Rate this list of product names for a pair of shoes that can fit any foot size. The rating should be out of 10, inline next to the product name:

```
UniFit SoleStrides
FlexiSize All-Fit Shoes
OmniFit FootFlex
...
AnyStep AdaptiWalkers
FlexiRange UniversalSteps
PerfectGlide MultiFit Shoes
```

Output:

```
UniFit SoleStrides: 7/10
FlexiSize All-Fit Shoes: 8/10
OmniFit FootFlex: 7/10
...
AnyStep AdaptiWalkers: 7/10
FlexiRange UniversalSteps: 7.5/10
PerfectGlide MultiFit Shoes: 9/10
```

This ranking is based on clarity, memorability, and how well the name communicates the unique selling point of the product (that they can fit any foot size). Remember, these ratings are subjective and the ultimate success of a product name also depends on the target market, branding, and marketing efforts.

In running this multiple times, it consistently rates the name “OneSize Glovewalkers” as the worst, providing context (if you ask) that the concept might be confusing in a shoe context. You may be wondering why, if the model *knows* this is a bad name, does it suggest it in the first place? LLMs work by predicting the next token in a sequence and therefore struggle to know what the overall response will be when

finished. However, when it has all the tokens from a previous response to review, it can more easily predict whether this would be labeled as a good or bad response.

We can continue to chain multiple calls together to improve the results of our task. For example, we could split this into three separate ratings: clarity, memorability, and how well the name communicates the unique selling point of the product. These ratings could then be given to a human as additional context on the final decision, or even calculated together to select the final name programmatically.

The real unlock in learning to work professionally with AI versus just playing around with prompting is realizing that every part of the system can be broken down into a series of iterative steps. Even with a single prompt this principles applies, as simply appending `Let's think step by step` to the prompt can lead to demonstrable gains in reasoning and proficiency, as well as provide an audit trail for quality assurance and debugging. When taking the time and tokens to reason, the ratings change and are more consistent with the scoring criteria.

Input:

`Let's think step by step. Rate this list of product names  
for a pair of shoes that can fit any foot size. The rating  
should be out of 10, inline next to the product name:`

`UniFit SoleStrides  
FlexiSize All-Fit Shoes  
OmniFit FootFlex  
...  
AnyStep AdaptiWalkers  
FlexiRange UniversalSteps  
PerfectGlide MultiFit Shoes`

Output:

`Here's a rating out of 10 for each product name, considering  
factors like descriptiveness, catchiness, and overall  
appeal:`

1. `**UniFit SoleStrides**: 7/10 - Sounds unifying but may not immediately convey the "fit any size" idea.`
2. `**FlexiSize All-Fit Shoes**: 8.5/10 - Combines flexibility and the notion that it fits all sizes.`
3. `**OmniFit FootFlex**: 8/10 - Omnipresence (fitting all) and flexibility nicely conveyed.`
- ...
18. `**AnyStep AdaptiWalkers**: 7.5/10 - "AnyStep" suggests versatility, but the overall name might be less direct.`
19. `**FlexiRange UniversalSteps**: 8/10 - Combines flexibility and universal fit.`
20. `**PerfectGlide MultiFit Shoes**: 8.5/10 - "MultiFit" is clear, and "PerfectGlide" suggests a comfortable experience.`

Remember, the best name can also be influenced by your target audience, branding strategy, and the overall message you want to convey.

OpenAI **calls this** “giving the model time to think,” and it is a key tenet of prompt engineering. In effect, *chain of thought* techniques like this, where the model is encouraged to list out its steps, are like dividing a task within the same prompt. Once we’ve automated product naming given a product idea, we can call ChatGPT again to describe each product, which in turn can be fed into Midjourney to generate an image of each product. Using an AI model to generate a prompt for an AI model is *meta prompting*, and it works because LLMs are human-level prompt engineers (Zhou, 2022).

Input:

Describe the new product "OneFit UltraStride Shoes," a shoe that can fit any foot size, in great detail:

Output:

Introducing the revolutionary "OneFit UltraStride Shoes," the footwear innovation designed to fit any foot size with unparalleled comfort and style. These shoes are a result of cutting-edge technology and meticulous craftsmanship, ensuring a truly personalized and enjoyable walking or running experience for everyone.

Design and Materials:

The OneFit UltraStride Shoes boast an elegant and modern design that appeals to both fashion-forward individuals and sports enthusiasts. They feature a sleek and streamlined silhouette, crafted using premium materials for durability and aesthetic appeal. The shoe's upper is constructed from a combination of breathable mesh and flexible synthetic materials, allowing for excellent airflow and a lightweight feel.

...

Color and Customization:

OneFit UltraStride Shoes are available in a variety of vibrant colors and stylish patterns, catering to diverse personal preferences. You can choose from classic monochromatic options for a timeless look or opt for bold and vibrant hues to make a fashion statement. The ability to customize the shoe's appearance allows individuals to express their unique style while enjoying the benefits of the OneFit technology.

Conclusion:

The OneFit UltraStride Shoes represent a breakthrough in footwear technology, eliminating the hassle of finding the

perfect fit for your feet. With their adaptive design, superior comfort, and stylish aesthetics, these shoes offer a truly personalized experience for individuals of all foot sizes. Stay comfortable, confident, and on-trend with the OneFit UltraStride Shoes, the ultimate choice for those seeking unmatched versatility and a perfect fit.

DALL-E is well-known by GPT-4, and therefore you can invoke its name within ChatGPT and it does a reasonable job at crafting a prompt for an image generation tool. If you were planning on using this prompt in production, you may consider applying the prompting principle of providing examples, but it does a good enough job for our purposes without examples.

Input:

Convert this description into a short prompt for an AI image generation model such as DALL-E:

Output:

Create an eye-catching image that visually represents the "OneFit UltraStride Shoes," a revolutionary footwear product that adapts to any foot size. Showcase a sleek and modern shoe design incorporating a stretchable upper, an adjustable lacing system, and a cushioned insole for optimal comfort. Emphasize the shoe's versatility by displaying it in action on various surfaces, highlighting its excellent traction. Incorporate vibrant colors and patterns to convey the customizable nature of the product. The image should capture the essence of innovation, style, and personalized fit that the OneFit UltraStride Shoes offer.

The output of this prompt can now be plugged into image generation tools like DALL-E or Midjourney as a prompt, which can give you a good starting point for visualizing what the product might look like. Although this might not be the final design you go with, seeing an image is more evocative and helps people form an opinion faster. It's easier cognitively to criticize or compliment an existing image than it is to imagine a new image from a blank page or section of text.

Figure 1-15 shows the output.

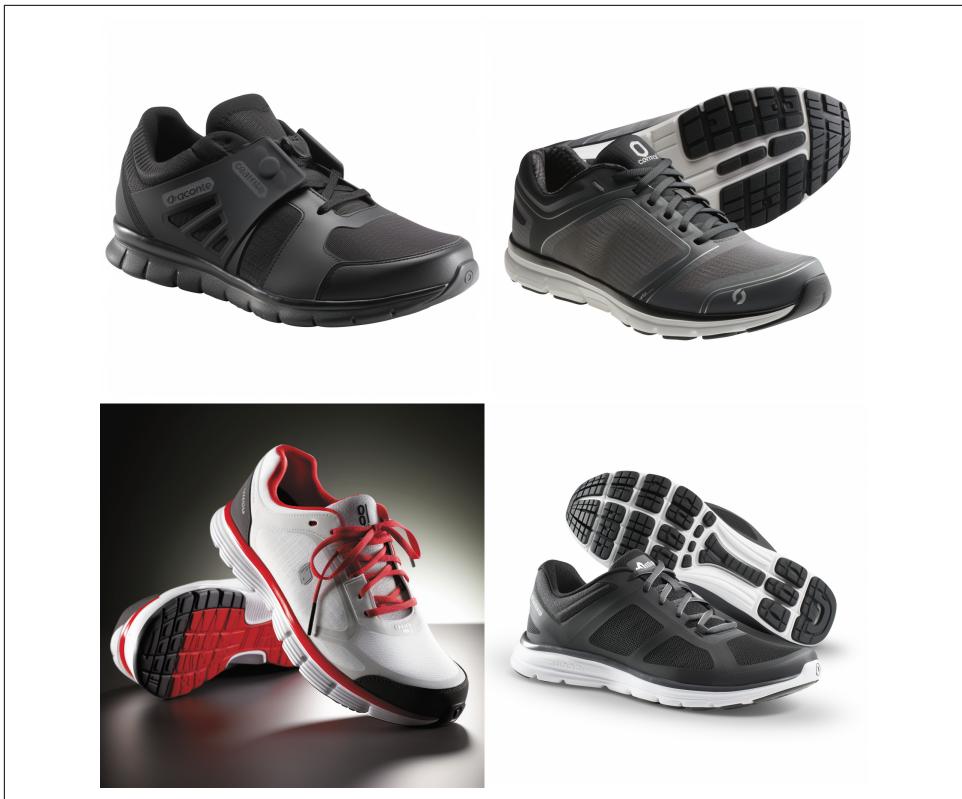


Figure 1-15. OneFit UltraStride shoes

It's common practice when working with AI professionally to chain multiple calls to AI together, and even multiple models, to accomplish more complex goals. Even single-prompt applications are often built dynamically, based on outside context queried from various databases or other calls to an AI model. The library [LangChain](#) has developed tooling for chaining multiple prompt templates and queries together, making this process more observable and well structured. A foundational example is progressive summarization, where text that is too large to fit into a context window can be split into multiple chunks of text, with each being summarized, before finally summarizing the summaries. If you talk to builders of early AI products, you'll find they're all under the hood chaining multiple prompts together, called *AI chaining*, to accomplish better results in the final output.

The [Reason and Act \(ReAct\)](#) framework was one of the first popular attempts at AI agents, including the open source projects [BabyAGI](#), [AgentGPT](#) and [Microsoft AutoGen](#). In effect, these agents are the result of chaining multiple AI calls together in order to plan, observe, act, and then evaluate the results of the action. Autonomous agents will be covered in [Chapter 6](#) but are still not widely used in production at the

time of writing. This practice of self-reasoning agents is still early and prone to errors, but there are promising signs this approach can be useful in achieving complex tasks, and is likely to be part of the next stage in evolution for AI systems.

There is an AI battle occurring between large tech firms like Microsoft and Google, as well as a wide array of open source projects on Hugging Face, and venture-funded start-ups like OpenAI and Anthropic. As new models continue to proliferate, they're diversifying in order to compete for different segments of the growing market. For example, Anthropic's Claude 2 had an [100,000-token context window](#), compared to GPT-4's standard [8,192 tokens](#). OpenAI soon responded with a [128,000-token window version of GPT-4](#), and Google touts a 1 million token context length with [Gemini 1.5](#). For comparison, one of the Harry Potter books would be around 185,000 tokens, so it may become common for an entire book to fit inside a single prompt, though processing millions of tokens with each API call may be cost prohibitive for most use cases.

This book focuses on GPT-4 for text generation techniques, as well as Midjourney v6 and Stable Diffusion XL for image generation techniques, but within months these models may no longer be state of the art. This means it will become increasingly important to be able to select the right model for the job and chain multiple AI systems together. Prompt templates are rarely comparable when transferring to a new model, but the effect of the Five Prompting Principles will consistently improve any prompt you use, for any model, getting you more reliable results.

## Summary

In this chapter, you learned about the importance of prompt engineering in the context of generative AI. We defined prompt engineering as the process of developing effective prompts that yield desired results when interacting with AI models. You discovered that providing clear direction, formatting the output, incorporating examples, establishing an evaluation system, and dividing complex tasks into smaller prompts are key principles of prompt engineering. By applying these principles and using common prompting techniques, you can improve the quality and reliability of AI-generated outputs.

You also explored the role of prompt engineering in generating product names and images. You saw how specifying the desired format and providing instructive examples can greatly influence the AI's output. Additionally, you learned about the concept of role-playing, where you can ask the AI to generate outputs as if it were a famous person like Steve Jobs. The chapter emphasized the need for clear direction and context to achieve desired outcomes when using generative AI models. Furthermore, you discovered the importance of evaluating the performance of AI models and the various methods used for measuring results, as well as the trade-offs between quality and token usage, cost, and latency.

In the next chapter, you will be introduced to text generation models. You will learn about the different types of foundation models and their capabilities, as well as their limitations. The chapter will also review the standard OpenAI offerings, as well as competitors and open source alternatives. By the end of the chapter, you will have a solid understanding of the history of text generation models and their relative strengths and weaknesses. This book will return to image generation prompting in Chapters 7, 8, and 9, so you should feel free to skip ahead if that is your immediate need. Get ready to dive deeper into the discipline of prompt engineering and expand your comfort working with AI.



# Introduction to Large Language Models for Text Generation

In artificial intelligence, a recent focus has been the evolution of large language models. Unlike their less-flexible predecessors, LLMs are capable of handling and learning from a much larger volume of data, resulting in the emergent capability of producing text that closely resembles human language output. These models have generalized across diverse applications, from writing content to automating software development and enabling real-time interactive chatbot experiences.

## What Are Text Generation Models?

Text generation models utilize advanced algorithms to understand the meaning in text and produce outputs that are often indistinguishable from human work. If you've ever interacted with ChatGPT or marveled at its ability to craft coherent and contextually relevant sentences, you've witnessed the power of an LLM in action.

In natural language processing (NLP) and LLMs, the fundamental linguistic unit is a *token*. Tokens can represent sentences, words, or even subwords such as a set of characters. A useful way to understand the size of text data is by looking at the number of tokens it comprises; for instance, a text of 100 tokens roughly equates to about 75 words. This comparison can be essential for managing the processing limits of LLMs as different models may have varying token capacities.

*Tokenization*, the process of breaking down text into tokens, is a crucial step in preparing data for NLP tasks. Several methods can be used for tokenization, including **Byte-Pair Encoding (BPE)**, WordPiece, and SentencePiece. Each of these methods has its unique advantages and is suited to particular use cases. BPE is commonly used due to its efficiency in handling a wide range of vocabulary while keeping the number of tokens manageable.

BPE begins by viewing a text as a series of individual characters. Over time, it combines characters that frequently appear together into single units, or tokens. To understand this better, consider the word *apple*. Initially, BPE might see it as *a*, *p*, *p*, *l*, and *e*. But after noticing that *p* often comes after *a* and before *l* in the dataset, it might combine them and treat *appl* as a single token in future instances.

This approach helps LLMs recognize and generate words or phrases, even if they weren't common in the training data, making the models more adaptable and versatile.

Understanding the workings of LLMs requires a grasp of the underlying mathematical principles that power these systems. Although the computations can be complex, we can simplify the core elements to provide an intuitive understanding of how these models operate. Particularly within a business context, the accuracy and reliability of LLMs are paramount.

A significant part of achieving this reliability lies in the pretraining and fine-tuning phases of LLM development. Initially, models are trained on vast datasets during the pretraining phase, acquiring a broad understanding of language. Subsequently, in the fine-tuning phase, models are adapted for specific tasks, honing their capabilities to provide accurate and reliable outputs for specialized applications.

## Vector Representations: The Numerical Essence of Language

In the realm of NLP, words aren't just alphabetic symbols. They can be tokenized and then represented in a numerical form, known as *vectors*. These vectors are multi-dimensional arrays of numbers that capture the semantic and syntactic relations:

$$w \rightarrow \mathbf{v} = [v_1, v_2, \dots, v_n]$$

Creating word vectors, also known as *word embeddings*, relies on intricate patterns within language. During an intensive training phase, models are designed to identify and learn these patterns, ensuring that words with similar meanings are mapped close to one another in a high-dimensional space ([Figure 2-1](#)).

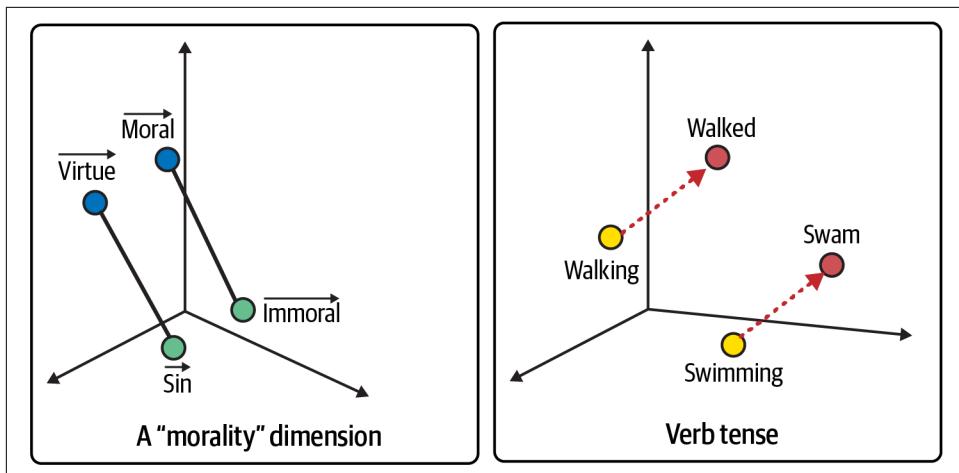


Figure 2-1. Semantic proximity of word vectors within a word embedding space

The beauty of this approach is its ability to capture nuanced relationships between words and calculate their distance. When we examine word embeddings, it becomes evident that words with similar or related meanings like *virtue* and *moral* or *walked* and *walking* are situated near each other. This spatial closeness in the embedding space becomes a powerful tool in various NLP tasks, enabling models to understand context, semantics, and the intricate web of relationships that form language.

## Transformer Architecture: Orchestrating Contextual Relationships

Before we go deep into the mechanics of transformer architectures, let's build a foundational understanding. In simple terms, when we have a sentence, say, *The cat sat on the mat*, each word in this sentence gets converted into its numerical vector representation. So, *cat* might become a series of numbers, as does *sat*, *on*, and *mat*.

As you'll explore in detail later in this chapter, the transformer architecture takes these word vectors and understands their relationships—both in structure (syntax) and meaning (semantics). There are many types of transformers; Figure 2-2 showcases both BERT and GPT's architecture. Additionally, a transformer doesn't just see words in isolation; it looks at *cat* and knows it's related to *sat* and *mat* in a specific way in this sentence.

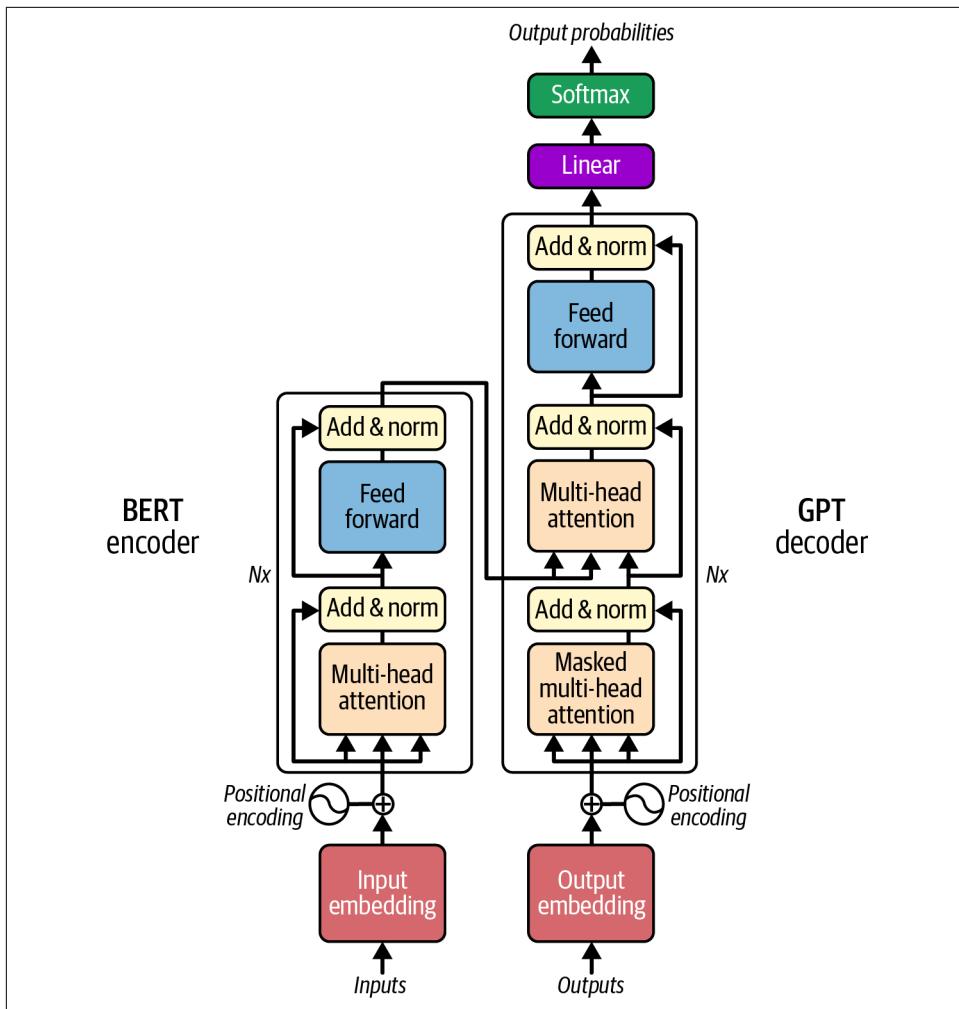


Figure 2-2. BERT uses an encoder for input data, while GPT has a decoder for output

When the transformer processes these vectors, it uses mathematical operations to understand the relationships between the words, thereby producing new vectors with rich, contextual information:

$$\mathbf{v}'_i = \text{Transformer}(\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m)$$

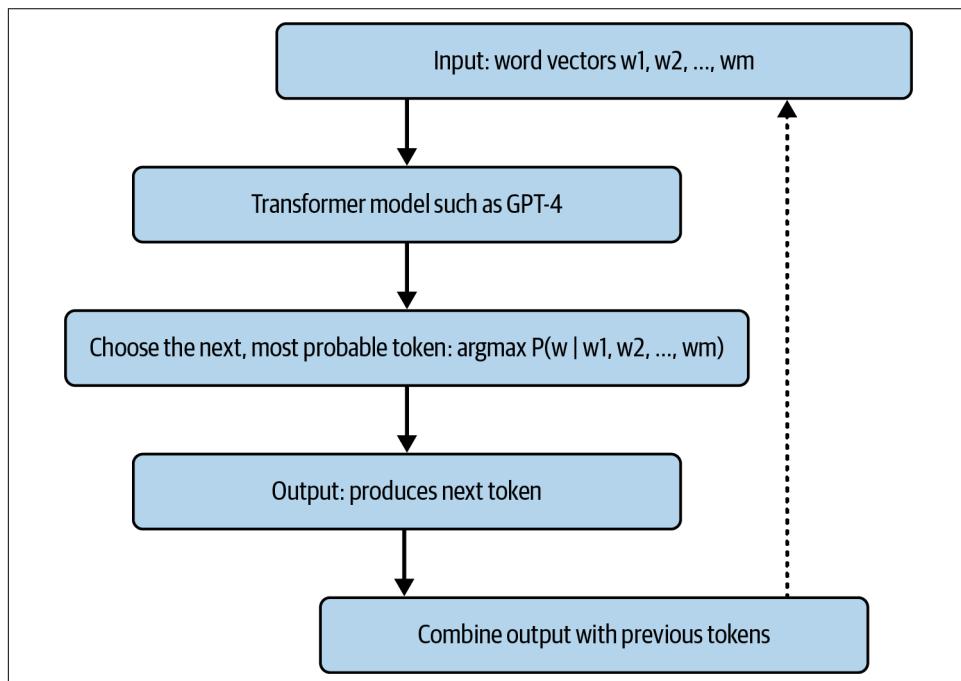
One of the remarkable features of transformers is their ability to comprehend the nuanced contextual meanings of words. The **self-attention** mechanism in transformers lets each word in a sentence look at all other words to understand its context better. Think of it like each word casting votes on the importance of other words for its meaning. By considering the entire sentence, transformers can more accurately determine the role and meaning of each word, making their *interpretations more contextually rich*.

## Probabilistic Text Generation: The Decision Mechanism

After the transformer understands the context of the given text, it moves on to generating new text, guided by the concept of likelihood or probability. In mathematical terms, the model calculates how likely each possible next word is to follow the current sequence of words and picks the one that is most likely:

$$w_{\text{next}} = \operatorname{argmax} P(w | w_1, w_2, \dots, w_m)$$

By repeating this process, as shown in [Figure 2-3](#), the model generates a coherent and contextually relevant string of text as its output.



*Figure 2-3. How text is generated using transformer models such as GPT-4*

The mechanisms driving LLMs are rooted in vector mathematics, linear transformations, and probabilistic models. While the under-the-hood operations are computationally intensive, the core concepts are built on these mathematical principles, offering a foundational understanding that bridges the gap between technical complexity and business applicability.

## Historical Underpinnings: The Rise of Transformer Architectures

Language models like ChatGPT (the *GPT* stands for *generative pretrained transformer*) didn't magically emerge. They're the culmination of years of progress in the field of NLP, with particular acceleration since the late 2010s. At the heart of this advancement is the introduction of transformer architectures, which were detailed in the groundbreaking paper "[Attention Is All You Need](#)" by the Google Brain team.

The real breakthrough of transformer architectures was the concept of *attention*. Traditional models processed text sequentially, which limited their understanding of language structure especially over long distances of text. Attention transformed this by allowing models to directly relate distant words to one another irrespective of their positions in the text. This was a groundbreaking proposition. It meant that words and their context didn't have to move through the entire model to affect each other. This not only significantly improved the models' text comprehension but also made them much more efficient.

This attention mechanism played a vital role in expanding the models' capacity to detect long-range dependencies in text. This was crucial for generating outputs that were not just contextually accurate and fluent, but also coherent over longer stretches.

According to AI pioneer and educator [Andrew Ng](#), much of the early NLP research, including the fundamental work on transformers, received significant funding from United States military intelligence agencies. Their keen interest in tools like machine translation and speech recognition, primarily for intelligence purposes, inadvertently paved the way for developments that transcended just translation.

Training LLMs requires extensive computational resources. These models are fed with vast amounts of data, ranging from terabytes to petabytes, including internet content, academic papers, books, and more niche datasets tailored for specific purposes. It's important to note, however, that the data used to train LLMs can carry *inherent biases from their sources*. Thus, users should exercise caution and ideally employ human oversight when leveraging these models, ensuring responsible and ethical AI applications.

OpenAI's GPT-4, for example, boasts an estimated **1.7 trillion parameters**, which is equivalent to an Excel spreadsheet that stretches across thirty thousand soccer fields. *Parameters* in the context of neural networks are the weights and biases adjusted throughout the training process, allowing the model to represent and generate complex patterns based on the data it's trained on. The training cost for GPT-4 was estimated to be in the order of **\$63 million**, and the training data would fill about **650 kilometers of bookshelves full of books**.

To meet these requirements, major technological companies such as Microsoft, Meta, and Google have invested heavily, making LLM development a high-stakes endeavor.

The rise of LLMs has provided an increased demand for the hardware industry, particularly companies specializing in graphics processing units (GPUs). NVIDIA, for instance, has become almost synonymous with high-performance GPUs that are essential for training LLMs.

The demand for powerful, efficient GPUs has skyrocketed as companies strive to build ever-larger and more complex models. It's not just the raw computational power that's sought after. GPUs also need to be fine-tuned for tasks endemic to machine learning, like tensor operations. *Tensors*, in a machine learning context, are multidimensional arrays of data, and operations on them are foundational to neural network computations. This emphasis on specialized capabilities has given rise to tailored hardware such as NVIDIA's H100 Tensor Core GPUs, explicitly crafted to expedite machine learning workloads.

Furthermore, the overwhelming demand often outstrips the supply of these top-tier GPUs, sending prices on an upward trajectory. This supply-demand interplay has transformed the GPU market into a fiercely competitive and profitable arena. Here, an eclectic clientele, ranging from tech behemoths to academic researchers, scramble to procure the most advanced hardware.

This surge in demand has sparked a wave of innovation beyond just GPUs. Companies are now focusing on creating dedicated AI hardware, such as Google's Tensor Processing Units (TPUs), to cater to the growing computational needs of AI models.

This evolving landscape underscores not just the symbiotic ties between software and hardware in the AI sphere but also spotlights the ripple effect of the LLM *gold rush*. It's steering innovations and funneling investments into various sectors, especially those offering the fundamental components for crafting these models.

# OpenAI's Generative Pretrained Transformers

Founded with a mission to ensure that artificial general intelligence benefits all of humanity, OpenAI has recently been at the forefront of the AI revolution. One of their most groundbreaking contributions has been the GPT series of models, which have substantially redefined the boundaries of what LLMs can achieve.

The original GPT model by OpenAI was more than a mere research output; it was a compelling demonstration of the potential of transformer-based architectures. This model showcased the initial steps toward making machines understand and generate human-like language, laying the foundation for future advancements.

The unveiling of GPT-2 was met with both anticipation and caution. Recognizing the model's powerful capabilities, OpenAI initially hesitated in releasing it due to concerns about its potential misuse. Such was the might of GPT-2 that ethical concerns took center stage, which might look quaint compared to the power of today's models. However, when OpenAI decided to release the project as [open-source](#), it didn't just mean making the code public. It allowed businesses and researchers to use these pretrained models as building blocks, incorporating AI into their applications without starting from scratch. This move democratized access to high-level natural language processing capabilities, spurring innovation across various domains.

After GPT-2, OpenAI decided to focus on releasing paid, closed-source models. GPT-3's arrival marked a monumental stride in the progression of LLMs. It garnered significant media attention, not just for its technical prowess but also for the societal implications of its capabilities. This model could produce text so convincing that it often became indistinguishable from human-written content. From crafting intricate pieces of literature to churning out operational code snippets, GPT-3 exemplified the seemingly boundless potential of AI.

## GPT-3.5-turbo and ChatGPT

Bolstered by Microsoft's significant investment in their company, OpenAI introduced GPT-3.5-turbo, an optimized version of its already exceptional predecessor. Following a [\\$1 billion injection](#) from Microsoft in 2019, which later increased to a hefty \$13 billion for a 49% stake in OpenAI's for-profit arm, OpenAI used these resources to develop GPT-3.5-turbo, which offered improved efficiency and affordability, effectively making LLMs more accessible for a broader range of use cases.

OpenAI wanted to gather more world feedback for fine-tuning, and so ChatGPT was born. Unlike its general-purpose siblings, ChatGPT was fine-tuned to excel in conversational contexts, enabling a dialogue between humans and machines that felt natural and meaningful.

Figure 2-4 shows the training process for ChatGPT, which involves three main steps:

#### *Collection of demonstration data*

In this step, human labelers provide examples of the desired model behavior on a distribution of prompts. The labelers are trained on the project and follow specific instructions to annotate the prompts accurately.

#### *Training a supervised policy*

The demonstration data collected in the previous step is used to fine-tune a pre-trained GPT-3 model using supervised learning. In supervised learning, models are trained on a labeled dataset where the correct answers are provided. This step helps the model to learn to follow the given instructions and produce outputs that align with the desired behavior.

#### *Collection of comparison data and reinforcement learning*

In this step, a dataset of model outputs is collected, and human labelers rank the outputs based on their preference. A reward model is then trained to predict which outputs the labelers would prefer. Finally, reinforcement learning techniques, specifically the Proximal Policy Optimization (PPO) algorithm, are used to optimize the supervised policy to maximize the reward from the reward model.

This training process allows the ChatGPT model to align its behavior with human intent. The use of reinforcement learning with human feedback helped create a model that is more helpful, honest, and safe compared to the pretrained GPT-3 model.

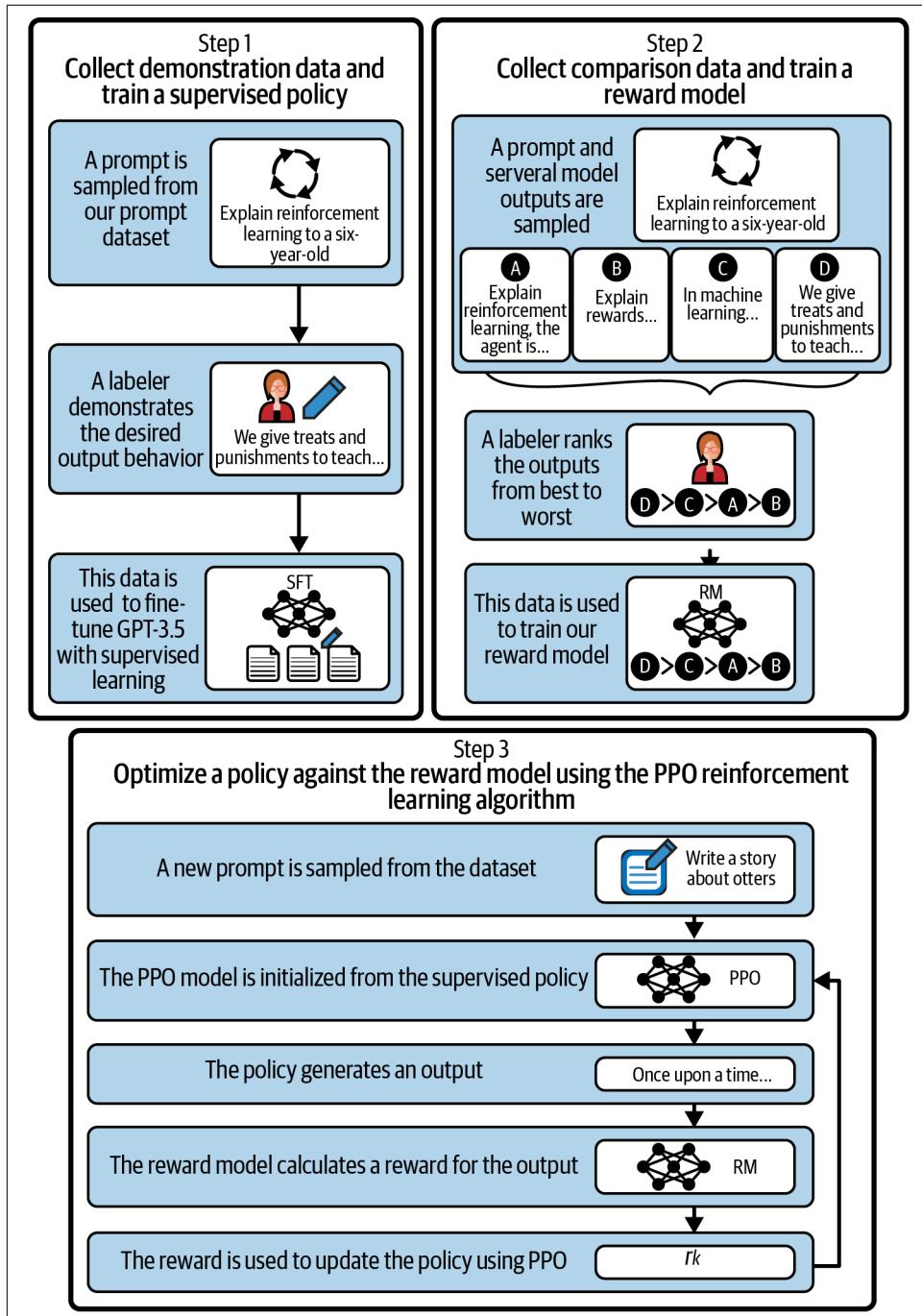


Figure 2-4. The fine-tuning process for ChatGPT

According to a [UBS study](#), by January 2023 ChatGPT set a new benchmark, amassing 100 million active users and becoming the fastest-growing consumer application in internet history. ChatGPT is now a go-to for customer service, virtual assistance, and numerous other applications that require the finesse of human-like conversation.

## GPT-4

In 2024, OpenAI released GPT-4, which excels in understanding complex queries and generating contextually relevant and coherent text. For example, GPT-4 scored in the 90th percentile of the bar exam with a score of 298 out of 400. Currently, GPT-3.5-turbo is free to use in ChatGPT, but GPT-4 requires a [monthly payment](#).

GPT-4 uses a [mixture-of-experts approach](#); it goes beyond relying on a single model's inference to produce even more accurate and insightful results.

On May 13, 2024, OpenAI introduced [GPT-4o](#), an advanced model capable of processing and reasoning across text, audio, and vision inputs in real time. This model offers enhanced performance, particularly in vision and audio understanding; it is also faster and more cost-effective than its predecessors due to its ability to process all three modalities in one neural network.

## Google's Gemini

After Google lost search market share due to ChatGPT usage, it initially released Bard on March 21, 2023. Bard was a bit [rough around the edges](#) and definitely didn't initially have the same high-quality LLM responses that ChatGPT offered ([Figure 2-5](#)).

Google has kept adding extra features over time including code generation, visual AI, real-time search, and voice into Bard, bringing it closer to ChatGPT in terms of quality.

On March 14, 2023, Google released [PaLM API](#), allowing developers to access it on Google Cloud Platform. In April 2023, Amazon Web Services (AWS) released similar services such as [Amazon Bedrock](#) and [Amazon's Titan FMs](#). Google [rebranded Bard to Gemini](#) for their v1.5 release in February 2024 and started to get results similar to GPT-4.

The screenshot shows a conversation between a user and Bard. The user asks, "What new discoveries from the James Space Webb Telescope can I tell my 9 year old about?" Bard responds with a blue circular icon and the text: "Your 9-year old might like these recent discoveries made by The James Webb Space Telescope (JWST):". Below this, there is a bulleted list of three facts:

- In 2023, The JWST discovered a number of galaxies nicknamed "green peas." They were given this name because they are small and round, like peas.
- The telescope captured images of galaxies that are over 13 billion years old. This means that the light from these galaxies has been traveling for over 13 billion years to reach us.
- JWST took the very first pictures of a planet outside of our own solar system. These distant worlds are called "exoplanets". Exo means "from outside".

Below the list, a note says, "These discoveries can spark a child's imagination about the infinite wonders of the universe". At the bottom of the interface are four interaction icons: thumbs up, thumbs down, a refresh arrow, and a 'Check it' button, followed by a vertical ellipsis.

Figure 2-5. Bard hallucinating results about the James Webb Space Telescope

Also, Google released two smaller **open source models** based on the same architecture as Gemini. OpenAI is finally no longer the only obvious option for software engineers to integrate state-of-the-art LLMs into their applications.

## Meta's Llama and Open Source

Meta's approach to language models differs significantly from other competitors in the industry. By sequentially releasing open source models **Llama**, **Llama 2** and **Llama 3**, Meta aims to foster a more inclusive and collaborative AI development ecosystem.

The open source nature of Llama 2 and Llama 3 has significant implications for the broader tech industry, especially for large enterprises. The transparency and collaborative ethos encourage rapid innovation, as problems and vulnerabilities can be quickly identified and addressed by the global developer community. As these models become more robust and secure, large corporations can adopt them with increased confidence.

Meta's open source strategy not only democratizes access to state-of-the-art AI technologies but also has the potential to make a meaningful impact across the industry. By setting the stage for a collaborative, transparent, and decentralized development process, Llama 2 and Llama 3 are pioneering models that could very well define the future of generative AI. The models are available in 7, 8 and 70 billion parameter versions on AWS, Google Cloud, Hugging Face, and other platforms.

The open source nature of these models presents a double-edged sword. On one hand, it levels the playing field. This means that even smaller developers have the

opportunity to contribute to innovation, improving and applying open source models to practical business applications. This kind of decentralized innovation could lead to breakthroughs that might not occur within the walled gardens of a single organization, enhancing the models' capabilities and applications.

However, the same openness that makes this possible also poses potential risks, as it could allow malicious actors to exploit this technology for detrimental ends. This indeed is a concern that organizations like OpenAI share, suggesting that some degree of control and restriction can actually serve to mitigate the dangerous applications of these powerful tools.

## Leveraging Quantization and LoRA

One of the game-changing aspects of these open source models is the potential for **quantization** and the use of **LoRA** (low-rank approximations). These techniques allow developers to fit the models into smaller hardware footprints. Quantization helps to reduce the numerical precision of the model's parameters, thereby shrinking the overall size of the model without a significant loss in performance. Meanwhile, LoRA assists in optimizing the network's architecture, making it more efficient to run on consumer-grade hardware.

Such optimizations make fine-tuning these LLMs increasingly feasible on consumer hardware. This is a critical development because it allows for greater experimentation and adaptability. No longer confined to high-powered data centers, individual developers, small businesses, and start-ups can now work on these models in more resource-constrained environments.

## Mistral

Mistral 7B, a brainchild of French start-up **Mistral AI**, emerges as a powerhouse in the generative AI domain, with its 7.3 billion parameters making a significant impact. This model is not just about size; it's about efficiency and capability, promising a bright future for open source large language models and their applicability across a myriad of use cases. The key to its efficiency is the implementation of sliding window attention, a technique released under a permissive Apache open source license. Many AI engineers have fine-tuned on top of this model as a base, including the impressive **Zephry 7b beta** model. There is also **Mixtral 8x7b**, a mixture of experts model (similar to the architecture of GPT-4), which achieves results similar to GPT-3.5-turbo.

For a more detailed and up-to-date comparison of open source models and their performance metrics, visit the Chatbot **Arena Leaderboard** hosted by Hugging Face.

## Anthropic: Claude

Released on July 11, 2023, **Claude 2** is setting itself apart from other prominent LLMs such as ChatGPT and LLaMA, with its pioneering **Constitutional AI** approach to AI safety and alignment—training the model using a list of rules or values. A notable enhancement in Claude 2 was its expanded context window of 100,000 tokens, as well as the ability to upload files. In the realm of generative AI, a *context window* refers to the amount of text or data the model can actively consider or keep in mind when generating a response. With a larger context window, the model can understand and generate based on a broader context.

This advancement garnered significant enthusiasm from AI engineers, as it opened up avenues for new and more intricate use cases. For instance, Claude 2's augmented ability to process more information at once makes it adept at summarizing extensive documents or sustaining in-depth conversations. The advantage was short-lived, as OpenAI released their 128K version of GPT-4 only **six months later**. However, the fierce competition between rivals is pushing the field forward.

The next generation of Claude included **Opus**, the first model to rival GPT-4 in terms of intelligence, as well as Haiku, a smaller model that is lightning-fast with the competitive price of \$0.25 per million tokens (half the cost of GPT-3.5-turbo at the time).

## GPT-4V(ision)

In a significant leap forward, on September 23, 2023, OpenAI expanded the capabilities of GPT-4 with the introduction of Vision, enabling users to instruct GPT-4 to analyze images alongside text. This innovation was also reflected in the update to ChatGPT's interface, which now supports the inclusion of both images and text as user inputs. This development signifies a major trend toward *multimodal models*, which can seamlessly process and understand multiple types of data, such as images and text, within a single context.

## Model Comparison

The market for LLMs is dominated by OpenAI at the time of writing, with its state-of-the-art GPT-4 model widely considered to have a significant lead. The closest competitor is Anthropic, and there is widespread excitement at the potential of smaller open source models such as Llama and Mistral, particularly with respects to fine-tuning. Although commentators expect OpenAI to continue to deliver world-beating models in the future, as open source models get *good enough* at more tasks, AI workloads may shift toward local fine-tuned models. With advances in model performance and quantization (methods for trading off accuracy versus size and

compute cost), it may be possible to one day run LLMs on your mobile phone or other devices.

For now, the best way to get a sense for what the models are good at is to run the same prompt across multiple models and compare the responses. One thing that regularly stands out in our work is that GPT-4 is much better at following instructions, as is demonstrated in the following example where it was the only model to respond in the right format, with names that matched the examples (starting with the letter *i*), as desired.

Input:

```
Brainstorm a list of product names for a shoe that fits any foot size,  
in the style of Steve Jobs.
```

```
Return the results as a comma-separated list, in this format:
```

```
Product description: A shoe that fits any foot size
```

```
Product names: [list of three product names]
```

```
## Examples
```

```
Product description: A refrigerator that dispenses beer
```

```
Product names: iBarFridge, iFridgeBeer, iDrinkBeerFridge
```

```
Product description: A watch that can tell accurate time in space
```

```
Product names: iNaut, iSpace, iTime
```

```
Product description: A home milkshake maker
```

```
Product names: iShake, iSmoothie, iShake Mini
```

Output (GPT-4):

```
Product description: A shoe that fits any foot size
```

```
Product names: iFit, iShoe, iFlexible
```

Output (Claude 3):

```
Product description: A shoe that fits any foot size
```

```
Product names: iFit, iComfort, iSole
```

Output (Llama 3 70b):

```
Here is the list of product names for a shoe that fits any foot size,  
in the style of Steve Jobs:
```

```
Product description: A shoe that fits any foot size Product names:
```

```
iFit, OneSize, iWalkFree
```

## Summary

The journey of LLMs from their inception to their current advanced states is a tale of relentless innovation, collaboration, and intense competition. As these models

continue to evolve, they are likely to become even more integral parts of our daily lives, changing the way we interact with technology and even with each other.

By understanding the historical context and capabilities of these models, you can better appreciate the tools at our disposal for various applications, from prompt engineering to the development of intelligent virtual agents. It's important to note, however, that while these models offer expansive possibilities, data privacy remains a crucial concern. If these models use your data for retraining or fine-tuning, exercise caution and refrain from inputting sensitive information.

In the next chapter, you will learn all the basic prompt engineering techniques for working with text LLMs. You'll learn the essential skills needed to get the most out of powerful language models like GPT-4. Exciting insights and practical methods await you as you unlock the true potential of generative AI.

# Standard Practices for Text Generation with ChatGPT

Simple prompting techniques will help you to maximize the output and formats from LLMs. You'll start by tailoring the prompts to explore all of the common practices used for text generation.

## Generating Lists

Automatically generating lists is incredibly powerful and enables you to focus on higher-level tasks while GPT can automatically generate, refine, rerank, and deduplicate lists on your behalf.

Input:

Generate a list of Disney characters.

Output:

Sure, here is a list of some popular Disney characters:

1. Mickey Mouse
2. Minnie Mouse
- ...
30. Bagheera (The Jungle Book)

GPT-4 is perfectly capable of providing a list of characters. However, there are some pitfalls with this approach:

- GPT has decided to provide 30 examples as a numbered list, separated by \n characters. However, if your downstream Python code was expecting to split on bullet points, then you'll likely end up with undesirable results or a runtime error.
- GPT has provided preceding commentary; removing any preceding/succeeding commentary would make parsing the output easier.
- The list size wasn't controlled and was left to the language model.
- Some of the characters have the name of their corresponding film within brackets—for example, *Bagheera* (*The Jungle Book*)—and others don't. This makes names harder to extract because you would need to remove the movie titles.
- No filtering or selection has been applied to the LLM generation based on our desired result.

Following you'll find an optimized prompt.

Input:

```
Generate a bullet-point list of 5 male Disney characters.  
Only include the name of the character for each line.  
Never include the film for each Disney character.  
Only return the Disney characters, never include any commentary.
```

Below is an example list:

- \* Aladdin
- \* Simba
- \* Beast
- \* Hercules
- \* Tarzan

Output:

- \* Woody
- \* Buzz Lightyear
- \* Stitch
- \* Jack Sparrow
- \* Prince Charming



### Provide Examples

Simply rephrasing your prompt to include examples (*few-shot prompting*) can greatly impact the desired output.

By optimizing the prompt, you've achieved the following:

- Restricted the list to a fixed size of five
- Generated only male characters
- Correctly formatted the list with bullet points
- Removed any preceding commentary

Simple lists are fine for most tasks; however, they are less structured, and for some tasks it's beneficial to obtain nested data structures from a GPT-4 output.

Three typical data structures include:

- Nested text data (hierarchical lists)
- JSON
- YAML

## Hierarchical List Generation

Hierarchical lists are useful for when your desired output is nested. A good example of this would be a detailed article structure.

Input:

Generate a hierarchical and incredibly detailed article outline on:

What are the benefits of data engineering.

See an example of the hierarchical structure below:

Article Title: What are the benefits of digital marketing?

- \* Introduction
  - a. Explanation of digital marketing
  - b. Importance of digital marketing in today's business world
- \* Increased Brand Awareness
  - a. Definition of brand awareness
  - b. How digital marketing helps in increasing brand awareness

Output:

Article Title: What are the benefits of data engineering?

- \* Introduction
  - a. Explanation of data engineering
  - b. Importance of data engineering in today's data-driven world

...(10 sections later)...

- \* Conclusion
  - a. Importance of data engineering in the modern business world
  - b. Future of data engineering and its impact on the data ecosystem

To generate an effective article outline in the preceding output, you've included two key phrases:

### *Hierarchical*

To suggest that the article outline needs to produce a nested structure.

### *Incredibly detailed*

To guide the language model towards producing a larger output. Other words that you could include that have the same effect would be *very long* or by specifying a large number of subheadings, *include at least 10 top-level headings*.



Asking a language model for a fixed number of items doesn't guarantee the language model will produce the same length. For example, if you ask for 10 headings, you might receive only 8. Therefore, your code should either validate that 10 headings exist or be flexible to handle varying lengths from the LLM.

So you've successfully produced a hierarchical article outline, but how could you parse the string into structured data?

Let's explore [Example 3-1](#) using Python, where you've previously made a successful API call against OpenAI's GPT-4. Two regular expressions are used to extract the headings and subheadings from `openai_result`. The `re` module in Python is used for working with regular expressions.

#### *Example 3-1. Parsing a hierarchical list*

```
import re

# openai_result = generate_article_outline(prompt)
# Commented out to focus on a fake LLM response, see below:

openai_result = '''
* Introduction
    a. Explanation of data engineering
    b. Importance of data engineering in today's data-driven world
* Efficient Data Management
    a. Definition of data management
    b. How data engineering helps in efficient data management
* Conclusion
    a. Importance of data engineering in the modern business world
    b. Future of data engineering and its impact on the data ecosystem
'''
```

```

# Regular expression patterns
heading_pattern = r'\* (.+'
subheading_pattern = r'\s+[a-z]\. (.+)'

# Extract headings and subheadings
headings = re.findall(heading_pattern, openai_result)
subheadings = re.findall(subheading_pattern, openai_result)

# Print results
print("Headings:\n")
for heading in headings:
    print(f"\* {heading}")

print("\nSubheadings:\n")
for subheading in subheadings:
    print(f"\* {subheading}")

```

This code will output:

```

Headings:
- Introduction
- Efficient Data Management
- Conclusion

Subheadings:
- Explanation of data engineering
- Importance of data engineering in today's data-driven world
- Definition of data management
- How data engineering helps in efficient data management
- Importance of data engineering in the modern business world
- Future of data engineering and its impact on the data ecosystem

```

The use of regular expressions allows for efficient pattern matching, making it possible to handle variations in the input text, such as the presence or absence of leading spaces or tabs. Let's explore how these patterns work:

```
heading_pattern = r'\* (.+')
```

This pattern is designed to extract the main headings and consists of:

- \\* matches the asterisk (\*) symbol at the beginning of a heading. The backslash is used to escape the asterisk, as the asterisk has a special meaning in regular expressions (zero or more occurrences of the preceding character).
- A space character will match after the asterisk.
- (.+): matches one or more characters, and the parentheses create a capturing group. The . is a wildcard that matches any character except a newline, and the +

is a quantifier that means *one or more* occurrences of the preceding element (the dot, in this case).

By applying this pattern you can easily extract all of the main headings into a list without the asterisk.

```
subheading_pattern = r'\s+[a-z]\. (.+)
```

The subheading pattern will match all of the subheadings within the openai\_result string:

- \s+ matches one or more whitespace characters (spaces, tabs, and so on). The + means *one or more* occurrences of the preceding element (the \s, in this case).
- [a-z] matches a single lowercase letter from *a* to *z*.
- \. matches a period character. The backslash is used to escape the period, as it has a special meaning in regular expressions (matches any character except a newline).
- A space character will match after the period.
- (.+) matches one or more characters, and the parentheses create a capturing group. The . is a wildcard that matches any character except a newline, and the + is a quantifier that means *one or more* occurrences of the preceding element (the dot, in this case).

Additionally the `re.findall()` function is used to find all non-overlapping matches of the patterns in the input string and return them as a list. The extracted headings and subheadings are then printed.

So now you're able to extract headings and subheadings from hierarchical article outlines; however, you can further refine the regular expressions so that each heading is associated with corresponding subheadings.

In [Example 3-2](#), the regex has been slightly modified so that each subheading is attached directly with its appropriate subheading.

### *Example 3-2. Parsing a hierarchical list into a Python dictionary*

```
import re

openai_result = """
* Introduction
    a. Explanation of data engineering
    b. Importance of data engineering in today's data-driven world
* Efficient Data Management
    a. Definition of data management
```

```

    b. How data engineering helps in efficient data management
    c. Why data engineering is important for data management
* Conclusion
    a. Importance of data engineering in the modern business world
    b. Future of data engineering and its impact on the data ecosystem
"""

```

```

section_regex = re.compile(r"\* (.+)")
subsection_regex = re.compile(r"\s*([a-z]\..+")

result_dict = {}
current_section = None

for line in openai_result.split("\n"):
    section_match = section_regex.match(line)
    subsection_match = subsection_regex.match(line)

    if section_match:
        current_section = section_match.group(1)
        result_dict[current_section] = []
    elif subsection_match and current_section is not None:
        result_dict[current_section].append(subsection_match.group(1))

print(result_dict)

```

This will output:

```

{
    "Introduction": [
        "a. Explanation of data engineering",
        "b. Importance of data engineering in today's data-driven world"
    ],
    "Efficient Data Management": [
        "a. Definition of data management",
        "b. How data engineering helps in efficient data management"
    ],
    "Conclusion": [
        "a. Importance of data engineering in the modern business world",
        "b. Future of data engineering and its impact on the data ecosystem"
    ]
}

```

The section title regex, `r'\* (.+)'`, matches an asterisk followed by a space and then one or more characters. The parentheses capture the text following the asterisk and space to be used later in the code.

The subsection regex, `r'\s*([a-z]\..+)'`, starts with `\s*`, which matches zero or more whitespace characters (spaces or tabs). This allows the regex to match subsections with or without leading spaces or tabs. The following part, `([a-z]\..+)`, matches a lowercase letter followed by a period and then one or more characters. The parentheses capture the entire matched subsection text for later use in the code.

The `for` loop iterates over each line in the input string, `openai_result`. Upon encountering a line that matches the section title regex, the loop sets the matched title as the current section and assigns an empty list as its value in the `result_dict` dictionary. When a line matches the subsection regex, the matched subsection text is appended to the list corresponding to the current section.

Consequently, the loop processes the *input string line by line*, categorizes lines as section titles or subsections, and constructs the intended dictionary structure.

## When to Avoid Using Regular Expressions

As you work to extract more structured data from LLM responses, relying solely on regular expressions can make the control flow *become increasingly complicated*. However, there are other formats that can facilitate the parsing of structured data from LLM responses with ease. Two common formats are `.json` and `.yml` files.

## Generating JSON

Let's start by experimenting with some prompt design that will direct an LLM to return a JSON response.

Input:

```
Compose a very detailed article outline on "The benefits of learning code" with a JSON payload structure that highlights key points.
```

```
Only return valid JSON.
```

Here is an example of the JSON structure:

```
{  
    "Introduction": [  
        "a. Explanation of data engineering",  
        "b. Importance of data engineering in today's data-driven world"],  
    ...  
    "Conclusion": [  
        "a. Importance of data engineering in the modern business world",  
        "b. Future of data engineering and its impact on the data ecosystem"]  
}
```

Output:

```
{  
    "Introduction": [  
        "a. Overview of coding and programming languages",  
        "b. Importance of coding in today's technology-driven world"],  
    ...  
    "Conclusion": [  
        "a. Recap of the benefits of learning code",  
        "b. Final thoughts on the future of data engineering"]  
}
```

```
        "b. The ongoing importance of coding skills in the modern world"]  
    }
```



### Give Direction and Provide Examples

Notice that in the preceding prompt, you've provided direction on the type of task, the format, and an example JSON output.

Common errors that you'll encounter when working with JSON involve invalid payloads, or the JSON being wrapped within triple backticks (```), such as:

Output:

```
Sure here's the JSON:  
```json  
{"Name": "John Smith"} # valid payload  
{"Name": "John Smith", "some_key":} # invalid payload  
```
```

Ideally you would like the model to respond like so:

Output:

```
{"Name": "John Smith"}
```

This is important because with the first output, you'd have to split after json and then parse the exact part of the string that contained valid JSON. There are several points that are worth adding to your prompts to improve JSON parsing:

```
You must follow the following principles:  
* Only return valid JSON  
* Never include backtick symbols such as: `  
* The response will be parsed with json.loads(), therefore it must be valid JSON.
```

Now let's examine how you can parse a **JSON output with Python**:

```
import json  
  
# openai_json_result = generate_article_outline(prompt)  
  
openai_json_result = """  
{  
    "Introduction": [  
        "a. Overview of coding and programming languages",  
        "b. Importance of coding in today's technology-driven world"],  
    "Conclusion": [  
        "a. Recap of the benefits of learning code",  
        "b. The ongoing importance of coding skills in the modern world"]  
}  
"""  
  
parsed_json_payload = json.loads(openai_json_result)  
print(parsed_json_payload)
```

```
'''{'Introduction': ['a. Overview of coding and programming languages',  
"b. Importance of coding in today's technology-driven world"],  
'Conclusion': ['a. Recap of the benefits of learning code',  
'b. The ongoing importance of coding skills in the modern world']}'''
```

Well done, you've successfully parsed some JSON.

As showcased, structuring data from an LLM response is streamlined when requesting the response in valid JSON format. Compared to the previously demonstrated regular expression parsing, this method is less cumbersome and more straightforward.

So what could go wrong?

- The language model accidentally adds extra text to the response such as `json output:` and your application logic only handles for valid JSON.
- The JSON produced isn't valid and fails upon parsing (either due to the size or simply for not escaping certain characters).

Later on you will examine strategies to gracefully handle for such edge cases.

## YAML

.yml files are a structured data format that offer different benefits over .json:

*No need to escape characters*

YAML's indentation pattern eliminates the need for braces, brackets, and commas to denote structure. This can lead to cleaner and less error-prone files, as there's less risk of mismatched or misplaced punctuation.

*Readability*

YAML is designed to be human-readable, with a simpler syntax and structure compared to JSON. This makes it easier for you to create, read, and edit prompts, especially when dealing with complex or nested structures.

*Comments*

Unlike JSON, YAML supports comments, allowing you to add annotations or explanations to the prompts directly in the file. This can be extremely helpful when working in a team or when revisiting the prompts after some time, as it allows for better understanding and collaboration.

Input:

- Below you'll find the current yaml schema.
- You can update the quantities based on a User Query.
- Filter the User Query based on the schema below, if it doesn't match and there are no items left then return ` "No Items" `.
- If there is a partial match, then return only the items that are

within the schema below:

```
# schema:  
- item: Apple Slices  
  quantity: 5  
  unit: pieces  
- item: Milk  
  quantity: 1  
  unit: gallon  
- item: Bread  
  quantity: 2  
  unit: loaves  
- item: Eggs  
  quantity: 1  
  unit: dozen
```

User Query: "5 apple slices, and 2 dozen eggs."

Given the schema below, please return only a valid `.yml` based on the User Query. If there's no match, return `"No Items"`. Do not provide any commentary or explanations.

Output:

```
- item: Apple Slices  
  quantity: 5  
  unit: pieces  
- item: Eggs  
  quantity: 2  
  unit: dozen
```

Notice with the preceding example how an LLM is able to infer the correct `.yml` format from the User Query string.

Additionally, you've given the LLM an opportunity to either:

- Return a valid `.yml` response
- Return a filtered `.yml` response

If after filtering, there are no `.yml` items left, then return `No Items`.

## Filtering YAML Payloads

You might decide to use this same prompt for cleaning/filtering a `.yml` payload.

First, let's focus on a payload that contains both valid and invalid schema in reference to our desired schema. `Apple slices` fit the criteria; however, `Bananas` doesn't exist, and you should expect for the User Query to be appropriately filtered.

Input:

```
# User Query:  
- item: Apple Slices  
  quantity: 5  
  unit: pieces  
- item: Bananas  
  quantity: 3  
  unit: pieces
```

Output:

```
# Updated yaml list  
- item: Apple Slices  
  quantity: 5  
  unit: pieces
```

In the preceding example, you've successfully filtered the user's payload against a set criteria and have used the language model as a *reasoning engine*.

By providing the LLM with a set of instructions within the prompt, the response is closely related to what a human might do if they were manually cleaning the data.

The input prompt facilitates the delegation of more control flow tasks to a language learning model (LLM), tasks that would typically require coding in a programming language like Python or JavaScript.

Figure 3-1 provides a detailed overview of the logic applied when processing user queries by an LLM.

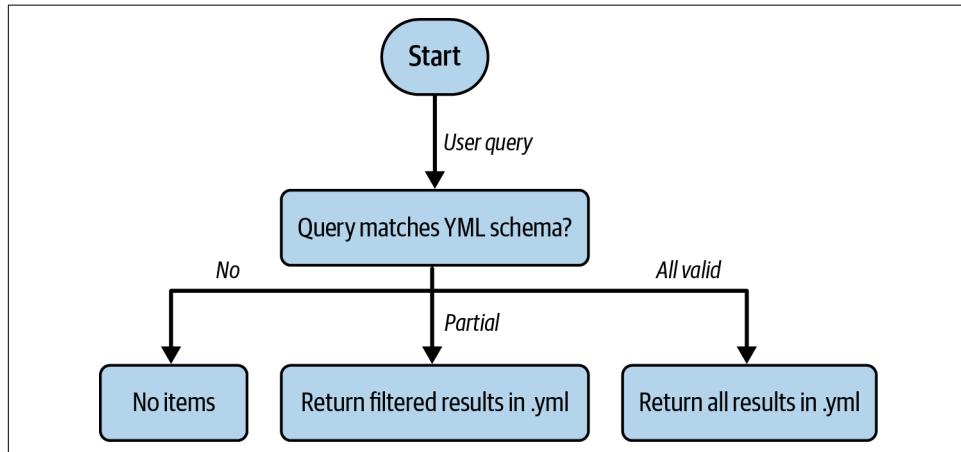


Figure 3-1. Using an LLM to determine the control flow of an application instead of code

# Handling Invalid Payloads in YAML

A completely invalid payload might look like this:

Input:

```
# User Query:  
- item: Bananas  
  quantity: 3  
  unit: pieces
```

Output:

```
No Items
```

As expected, the LLM returned `No Items` as none of the `User Query` items matched against the previously defined `schema`.

Let's create a Python script that gracefully accommodates for the various types of LLM results returned. The core parts of the script will focus on:

- Creating custom exceptions for each type of error that might occur due to the three LLM response scenarios
- Parsing the proposed schema
- Running a series of custom checks against the response so you can be sure that the YML response can be safely passed to downstream software applications/microservices

You could define six specific errors that would handle for all of the edge cases:

```
class InvalidResponse(Exception):  
    pass  
  
class InvalidItemType(Exception):  
    pass  
  
class InvalidItemKeys(Exception):  
    pass  
  
class InvalidItemName(Exception):  
    pass  
  
class InvalidItemQuantity(Exception):  
    pass  
  
class InvalidItemUnit(Exception):  
    pass
```

Then provide the previously proposed YAML schema as a string:

```
# Provided schema
schema = """
- item: Apple Slices
  quantity: 5
  unit: pieces
- item: Milk
  quantity: 1
  unit: gallon
- item: Bread
  quantity: 2
  unit: loaves
- item: Eggs
  quantity: 1
  unit: dozen
"""

```

Import the `yaml` module and create a custom parser function called `validate_response` that allows you to easily determine whether an LLM output is valid:

```
import yaml

def validate_response(response, schema):
    # Parse the schema
    schema_parsed = yaml.safe_load(schema)
    maximum_quantity = 10

    # Check if the response is a list
    if not isinstance(response, list):
        raise InvalidResponse("Response is not a list")

    # Check if each item in the list is a dictionary
    for item in response:
        if not isinstance(item, dict):
            raise InvalidItemType("Item is not a dictionary")

    # Check if each dictionary has the keys "item", "quantity", and "unit"
    if not all(key in item for key in ("item", "quantity", "unit")):
        raise InvalidItemKeys("Item does not have the correct keys")

    # Check if the values associated with each key are the correct type
    if not isinstance(item["item"], str):
        raise InvalidItemName("Item name is not a string")
    if not isinstance(item["quantity"], int):
        raise InvalidItemQuantity("Item quantity is not an integer")
    if not isinstance(item["unit"], str):
        raise InvalidItemUnit("Item unit is not a string")

    # Check if the values associated with each key are the correct value
    if item["item"] not in [x["item"] for x in schema_parsed]:
```

```

        raise InvalidItemName("Item name is not in schema")
    if item["quantity"] > maximum_quantity:
        raise InvalidItemQuantity(f"Item quantity is greater than
        {maximum_quantity}")
    if item["unit"] not in ["pieces", "dozen"]:
        raise InvalidItemUnit("Item unit is not pieces or dozen")

```

To test these edge cases, following you'll find several mocked LLM responses:

```

# Fake responses
fake_response_1 = """
- item: Apple Slices
  quantity: 5
  unit: pieces
- item: Eggs
  quantity: 2
  unit: dozen
"""

fake_response_2 = """
# Updated yaml list
- item: Apple Slices
  quantity: 5
  unit: pieces
"""

fake_response_3 = """Unmatched"""

```

Finally, now you can:

- Use `yaml.safe_load(response)` to safely parse the `.yml` schema
- Call the `validate_response` function for each LLM response to test it against custom `.yml` validation logic

```

# Parse the fake responses
response_1_parsed = yaml.safe_load(fake_response_1)
response_2_parsed = yaml.safe_load(fake_response_2)
response_3_parsed = yaml.safe_load(fake_response_3)

# Validate the responses against the schema
try:
    validate_response(response_1_parsed, schema)
    print("Response 1 is valid")
except Exception as e:
    print("Response 1 is invalid:", str(e))

try:
    validate_response(response_2_parsed, schema)
    print("Response 2 is valid")
except Exception as e:
    print("Response 2 is invalid:", str(e))

```

```

try:
    validate_response(response_3_parsed, schema)
    print("Response 3 is valid")
except Exception as e:
    print("Response 3 is invalid:", str(e))

Response 1 is valid
Response 2 is valid
Response 3 is invalid: Response is not a list

```

## Diverse Format Generation with ChatGPT

ChatGPT is not only capable of generating `.json` or `.yml` files but can also support many other data formats. For example, it can aid you in creating a mermaid diagram or generating a mock `.csv` document. Its versatility stretches to even generating code, conversations, and scripts.

Let's explore in detail how ChatGPT can produce diverse formatted content. Mermaid is a markdown-like scripting language for generating charts, diagrams, and other visualizations.

Input:

```

Create a flowchart diagram in mermaid syntax showing a simple food ordering
process that starts with "Choose food" and ends with "Pay for meal."
Include steps for "Add food to cart" and "Confirm cart."

```

Output:

```

graph TD
    ChooseFood[Choose Food] --> AddToCart[Add to Cart]
    AddToCart --> ConfirmCart[Confirm Cart]
    ConfirmCart --> PayForMeal[Pay for Meal]

```

This output, when input to a mermaid diagram renderer, will create a precise flowchart reflecting the food ordering process (Figure 3-2).

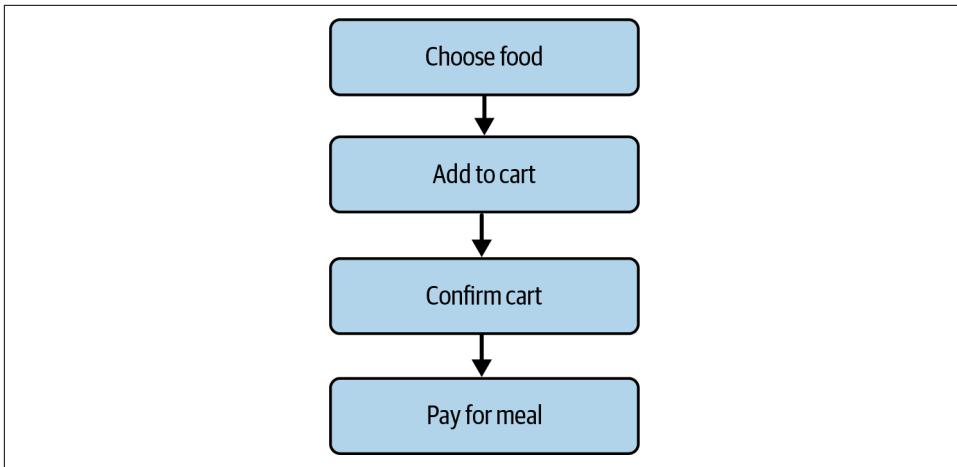


Figure 3-2. A streamlined flow diagram created using mermaid syntax via an LLM

## Mock CSV Data

ChatGPT can also generate mock CSV data that can be utilized for testing purposes or other uses.

Input:

```
Generate a sample CSV data for five students with the following fields:  
name, age, grade.
```

Output:

```
name,age,grade  
John,16,A  
Alex,17,B  
Emma,16,A  
Sophia,16,B  
Liam,17,A
```

The example generated is a well-formatted CSV file, which can be loaded directly into many tools that read CSV data.

Remember, GPT models rely heavily on the format you specify. For instance, specifying CSV implies the AI should use commas as separators. Be clear about the desired format for optimal results.

You've explored different ways of extracting structured data from language models including regular expressions, JSON, YML, and other formats. You can also now use LLMs to direct parts of your application's control flow.

## Explain It like I'm Five

The *Explain It like I'm Five* prompt focuses on explaining a given section of text *as if you were talking to a five-year-old child*.

This style of prompting is very effective and is a great way to convert technical documents into simple summaries that anyone can understand.

Let's apply this prompt to an abstract of "**Tumor Microenvironment: Recent Advances in Various Cancer Treatments**".

Input:

Explain the following text as if I was 5 years old. Explain it like I'm 5.

Text: This is a review regarding different types of cancer treatments. We aimed at analyzing the tumor microenvironment and the recent trends for the therapeutic applications and effectiveness for several kinds of cancers. Traditionally the cancer treatment was based on the neoplastic cells.

...(shortened for brevity)...

Output:

Cancer is a sickness that makes your body's cells grow and divide in a way that is not normal. Doctors have different ways to treat cancer, like cutting it out, giving you medicine, or using special rays to kill the cancer cells...

The ChatGPT output from the preceding example is much more digestible than the abstract.

## Universal Translation Through LLMs

Additionally, language models can potentially act as *universal translators* due to their proficiency in understanding and generating human-like text across multiple languages.

Let's investigate this with a simple example:

1. Generate some simple text.
2. Change the simple text and make it difficult to read.
3. Then transform the same text back into an easy to read format in Spanish (instead of English).
4. Revert the simplistic text back into English.

Input:

Create some simple text.

Output:

The sun gently rises, painting the sky with hues of orange and pink. Birds chirp merrily as they welcome the new day. The cool breeze whispers through the trees, carrying the scent of fresh grass and blooming flowers. Life awakens with a sense of tranquility and promise.

...(shortened for brevity)...

Input:

Now make the text incredibly complex and hard to read.

Output:

In the ethereal realm of celestial ascendancy, wherein the resplendent orb of the sun commences its gradual ascent, an exquisite phenomenon unfolds, engendering a symphony of chromatic brilliance upon the expansive canopy of the firmament. Avian denizens, endowed with mellifluous voices, bespeak their jubilant salutations, punctuating the aural tapestry with notes of felicity. The zephyr, tenderly wafting amidst arboreal sentinels, carries upon its gentle currents the redolence of verdant blades and blossoming flora.

...(shortened for brevity)...

Input:

Now transform the text into a simple, easy to read format that's in Spanish , make sure that it is very easy to understand and read. You must ensure that it is as simple to comprehend as possible.

- Use simple language.
- Make it concise.

Output:

En un lugar mágico y celestial, cuando el sol empieza a salir, ocurre algo hermoso en el cielo. Los pájaros cantan alegremente, llenando el aire con sus dulces melodías. Una brisa suave sopla entre los árboles, llevando consigo el olor fresco de la naturaleza. Todo cobra vida y promete momentos felices. Nos invita a disfrutar de las cosas simples y a encontrar la felicidad en los pequeños detalles. En este escenario encantador, encontramos paz y aprendemos que hasta los momentos más simples pueden ser mágicos.

Input:

Finally translate it back into English.

Output:

In a magical and celestial place, when the sun begins to rise, something beautiful happens in the sky. The birds sing cheerfully, filling the air with their sweet melodies. A gentle breeze blows among the trees, carrying the fresh scent of nature. Everything comes alive and promises happy moments. It invites us to enjoy the simple things and find happiness in the little details. In this enchanting setting, we find peace and learn that even the simplest moments can be magical.

As you can see, part of the meaning is lost while doing the translation, and this is to be expected. However, large parts of the original text's intent remains, demonstrating that LLMs can act as *highly effective translators* for languages that have adequate resources on the internet. For languages that do not have a good amount of resources, it will yield bad results.

The same logic applies to coding languages. LLMs are very good at generating code for established programming languages such as Python and JavaScript but perform worse for newer coding languages and packages.

The boundaries between different forms of information are becoming *increasingly fluid*. The essence of information itself is evolving, allowing for effortless transformations of summaries into stories, poems, or other creative expressions, ultimately enriching our understanding and engagement with the content.

*Diffusion models* are a unique class of generative models utilized in machine learning, specifically designed to produce new images that mimic those found in the training set.

Moreover, when you combine language models with diffusion models, it enables seamless transitions between text, video, and other modalities. This makes it even simpler for you to convey complex ideas across various formats, facilitating a more accessible and comprehensive experience.

## Ask for Context

LLMs are not only capable of generating text but can also act as simple agents with a limited amount of *reasoning capability*. This allows you to write a prompt asking the language model to either:

- Return a valid result to a question or statement
- Ask for more context to appropriately answer the question

In this section, you'll learn about the importance of *asking for context* when working with LLMs such as GPT-4. We will start with an example of a prompt that doesn't provide enough context, resulting in a less useful response.

Then, we will provide a better prompt that encourages the model to ask for additional context if needed. Finally, we will use the additional context provided to generate a more informed response.

Input:

Should I use MongoDB or PostgreSQL for my project?

Output:

It's difficult to make a recommendation without knowing more about your specific software requirements. Both MongoDB and PostgreSQL have their advantages, but the best choice depends on the specific needs of your project.

In the preceding prompt, the model does not have enough context to make a meaningful recommendation. Instead, you can ask ChatGPT for a list of recommended points that would help it to make an effective decision.

Input:

Should I use MongoDB or PostgreSQL for my project?

Provide a list of requirements that would help you to make an informed and high-quality decision about which database solution I should choose?

Output:

To make an informed decision between MongoDB and PostgreSQL for your project, consider the following requirements:

1. Data Structure: Evaluate the structure of your data. If your data is highly unstructured or schema-less, MongoDB's flexible document model may be more suitable. On the other hand, if your data has a fixed schema or requires complex relationships, PostgreSQL's relational model might be a better fit.

...

10. Cost: Assess the cost implications of each database solution. MongoDB offers both a community edition (free) and a commercial edition with additional features and support. PostgreSQL is open-source and free to use, with optional commercial support available from various vendors.

After prompting ChatGPT about how to make an informed decision, now you're aware of what to include within your prompt to help the language model with deciding.

Input:

Should I use MongoDB or PostgreSQL for my project?

Context:

- Data Structure: Highly unstructured data with varying document formats and no fixed schema. Scalability: Project requires the ability to handle massive amounts of data and distribute it across multiple servers.
  - ACID Compliance: Strong data consistency and integrity are crucial for the project's financial transactions.
  - Querying Capabilities: Complex geospatial queries and aggregations on location-based data.
- Community and Ecosystem: Active community with numerous third-party libraries and tools specifically tailored for the project's industry.
- Data Integrity and Constraints: Enforcing strict foreign key relationships and complex data validation rules.
  - Transactions and Concurrency: Requires handling high levels of concurrent read and write operations while maintaining data consistency.
  - Maturity and Stability: Prioritizing a well-established and extensively tested database for long-term stability.
  - Development and Operational Ease: Agile development approach with a need for rapid prototyping and iterative schema design.
  - Cost: Budget constraints necessitate using an open-source solution with no additional licensing or support costs.

If you need more context, please specify what would help you to make a better decision.

Output:

Based on these considerations, both MongoDB and PostgreSQL have their strengths, but given the emphasis on strong data consistency, complex querying, and data integrity constraints, PostgreSQL seems to be a more suitable choice for your project.

In this final example, the model uses the additional context provided to give a well-informed recommendation for using PostgreSQL. By asking for context when necessary, LLMs like ChatGPT and GPT-4 can deliver more valuable and accurate responses.

**Figure 3-3** demonstrates how *asking for context* changes the decision-making process of LLMs. Upon receiving user input, the model first assesses whether the context given is sufficient. If not, it prompts the user to provide more detailed information, emphasizing the model's reliance on context-rich inputs. Once adequate context is acquired, the LLM then generates an informed and relevant response.

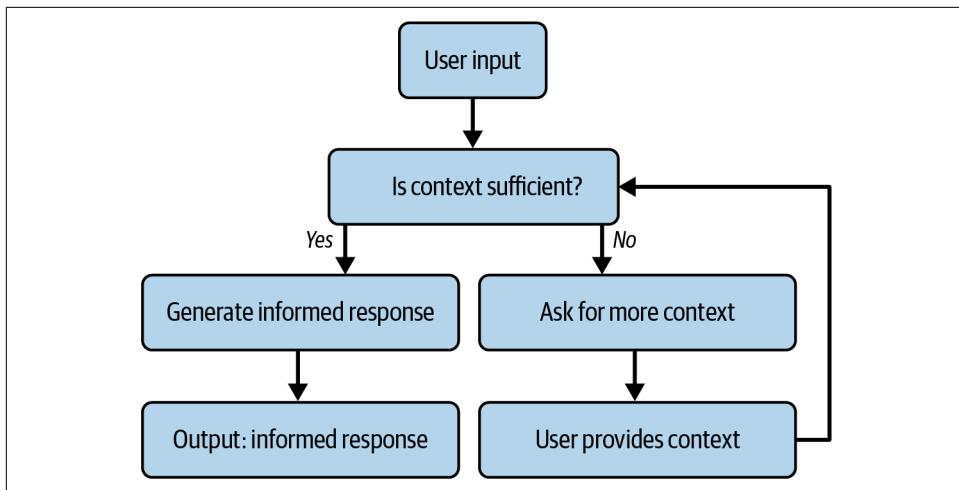


Figure 3-3. The decision process of an LLM while asking for context



### Allow the LLM to Ask for More Context by Default

You can allow the LLM to ask for more context as a default by including this key phrase: *If you need more context, please specify what would help you to make a better decision.*

In this section, you've seen how LLMs can act as agents that use environmental context to make decisions. By iteratively refining the prompt based on the model's recommendations, we eventually reach a point where the model has *enough context to make a well-informed decision*.

This process highlights the importance of providing sufficient context in your prompts and being prepared to ask for more information when necessary. By doing so, you can leverage the power of LLMs like GPT-4 to make more accurate and valuable recommendations.

In agent-based systems like GPT-4, the ability to ask for more context and provide a finalized answer is crucial for making well-informed decisions. [AutoGPT](#), a multi-agent system, has a self-evaluation step that automatically checks whether the task can be completed given the current context within the prompt. This technique uses an actor-critic relationship, where the existing prompt context is being analyzed to see whether it could be further refined before being executed.

# Text Style Unbundling

*Text style unbundling* is a powerful technique in prompt engineering that allows you to extract and isolate specific textual features from a given document, such as tone, length, vocabulary, and structure.

This allows you to create new content that shares similar characteristics with the original document, ensuring consistency in style and tone across various forms of communication.

This consistency can be crucial for businesses and organizations that need to communicate with a unified voice across different channels and platforms. The benefits of this technique include:

## *Improved brand consistency*

By ensuring that all content follows a similar style, organizations can strengthen their brand identity and maintain a cohesive image.

## *Streamlined content creation*

By providing a clear set of guidelines, writers and content creators can more easily produce materials that align with a desired style.

## *Adaptability*

Text style unbundling allows for the easy adaptation of existing content to new formats or styles while preserving the core message and tone.

The process of text style unbundling involves *identifying the desired textual features* or creating a meta prompt (a prompt to create prompts) to extract these features and then using the extracted features to guide the generation of new content.

## Identifying the Desired Textual Features

To successfully unbundle a text style, you must first identify the specific features you want to extract from the input document. Common textual features to consider include:

### *Tone of voice*

The overall mood or attitude conveyed by the text, such as formal, casual, humorous, or authoritative

### *Length*

The desired word count or general length of the content

### *Vocabulary and phrasing*

The choice of words and expressions, including industry-specific jargon, colloquialisms, and complexity

Additionally, you can even create a meta prompt that will dynamically show you these.

Input:

Analyze the following text and identify core features that will allow further documents to imitate the same style, consistency, and formatting as the document below.

Provide all of the features as a style writing guide that will be used for future documents.

Analyze the following text:

Artificial intelligence is rapidly transforming the world around us, opening up new possibilities and revolutionizing industries. From health care to entertainment, AI is making our lives better and more convenient in countless ways. However, with great power comes great responsibility. We must ensure that AI technologies are developed and deployed ethically and responsibly to protect our privacy and uphold our values.

Output:

Style Writing Guide:

Tone of Voice

- Maintain an informative and optimistic tone throughout the document.
- Emphasize the positive impact of the subject matter while acknowledging any potential concerns or responsibilities.

Length:

...

Vocabulary:

...

Structure:

...

Content:

- Focus on the transformative nature and positive impacts of the subject matter.
- Highlight specific industries or areas where the subject matter has made a difference.
- Acknowledge any associated responsibilities or potential risks, emphasizing the need for ethical and responsible practices.

Stress the importance of privacy protection and upholding values in relation to the subject matter.

# Generating New Content with the Extracted Features

After you've extracted the desired features from the input document, you can use this information to guide future content generation. Remember to ensure that the AI model incorporates the extracted features into its output by specifying the desired style in your prompt. For example:

```
Write a new blog post on [topic] using the same tone of voice,  
length, vocabulary, and structure as the previously analyzed text.
```

By combining this technique with *reference text* (documents that act as grounding truth), you can produce credible, branded content that requires minimal revisions.

## Extracting Specific Textual Features with LLMs

You can easily tailor a prompt to guide an LLM in extracting particular textual features from a document. This can be applied beyond just analyzing text for copywriting purposes. For instance, recognizing entities or discerning sentiment from the text can be achieved by crafting a precise instruction for the LLM.

Input:

Analyze the following text to identify and list the entities mentioned:

Artificial intelligence is rapidly transforming the world around us, opening up new possibilities and revolutionizing industries. From health care to entertainment, AI is making our lives better and more convenient in countless ways. However, with great power comes great responsibility. We must ensure that AI technologies are developed and deployed ethically and responsibly to protect our privacy and uphold our values.

Output:

Entities:

- Artificial Intelligence (AI)
- Health care
- Entertainment
- Privacy
- Values

## Summarization

In the era of information overload, the ability to condense large amounts of text into concise, digestible summaries has become an essential skill. As the volume of data we produce and consume continues to grow, the demand for effective summarization techniques has increased significantly.

AI has made significant advancements in this domain, providing powerful tools for generating summaries that retain the most critical information while discarding less relevant details.

Let's explore how to summarize using a language model.

Input:

Summarize the following text:

Whereas the party of the first part, hereinafter referred to as the "Purchaser," and the party of the second part, hereinafter referred to as the "Vendor," being fully cognizant of their respective legal obligations and responsibilities pursuant to the applicable statutory provisions and common law doctrines, have duly entered into a legally binding agreement, hereinafter referred to as the "Agreement," for the purpose of effectuating a transaction involving the transfer of goods, hereinafter referred to as the "Goods," between the Purchaser and the Vendor, subject to the terms and conditions set forth herein, the Parties do hereby solemnly acknowledge and affirm the following:

...

Output:

This is a legally binding agreement between the Purchaser and the Vendor for the transfer of goods. The agreement incorporates the recitals as an integral part, but in case of any conflict, the operative provisions of the agreement prevail. The interpretation clause states that singular words include the plural and vice versa, and words referring to a gender include all genders. References to statutory provisions include any amendments or replacements.

Summarization is an invaluable application of AI, enabling users to quickly extract key insights from lengthy articles, reports, or research papers. This process can help individuals make informed decisions, save time, and prioritize their reading. AI-generated summaries can also facilitate information sharing among teams, allowing for more efficient collaboration and communication.

## Summarizing Given Context Window Limitations

For documents larger than an LLM can handle in a single API request, a common approach is to chunk the document, summarize each chunk, and then combine these summaries into a final summary, as shown in [Figure 3-4](#).

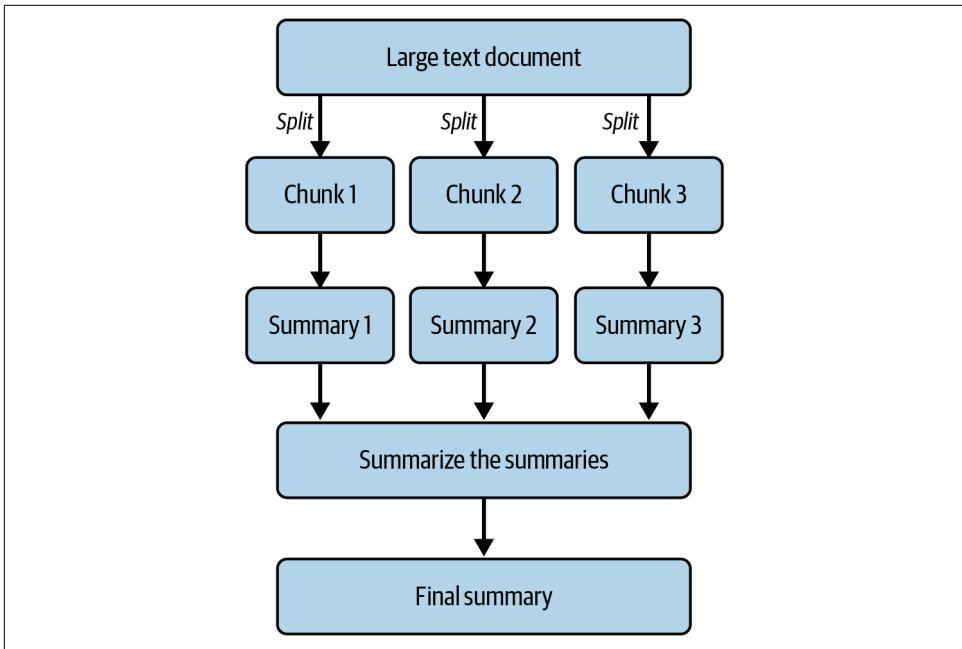


Figure 3-4. A summarization pipeline that uses text splitting and multiple summarization steps

Additionally, people may require different types of summaries for various reasons, and this is where AI summarization comes in handy. As illustrated in the preceding diagram, a large PDF document could easily be processed using AI summarization to generate distinct summaries tailored to individual needs:

#### *Summary A*

Provides key insights, which is perfect for users seeking a quick understanding of the document's content, enabling them to focus on the most crucial points

#### *Summary B*

On the other hand, offers decision-making information, allowing users to make informed decisions based on the content's implications and recommendations

#### *Summary C*

Caters to collaboration and communication, ensuring that users can efficiently share the document's information and work together seamlessly

By customizing the summaries for different users, AI summarization contributes to increased information retrieval for all users, making the entire process more efficient and targeted.

Let's assume you're only interested in finding and summarizing information about the advantages of digital marketing. Simply change your summarization prompt to **Provide a concise, abstractive summary of the above text. Only summarize the advantages: ...**

AI-powered summarization has emerged as an essential tool for quickly distilling vast amounts of information into concise, digestible summaries that cater to various user needs. By leveraging advanced language models like GPT-4, AI summarization techniques can efficiently extract key insights and decision-making information, and also facilitate collaboration and communication.

As the volume of data continues to grow, the demand for effective and targeted summarization will only increase, making AI a crucial asset for individuals and organizations alike in navigating the Information Age.

## Chunking Text

LLMs continue to develop and play an increasingly crucial role in various applications, as the ability to process and manage large volumes of text becomes ever more important. An essential technique for handling large-scale text is known as *chunking*.

*Chunking* refers to the process of breaking down large pieces of text into smaller, more manageable units or chunks. These chunks can be based on various criteria, such as sentence, paragraph, topic, complexity, or length. By dividing text into smaller segments, AI models can more efficiently process, analyze, and generate responses.

**Figure 3-5** illustrates the process of chunking a large piece of text and subsequently extracting topics from the individual chunks.

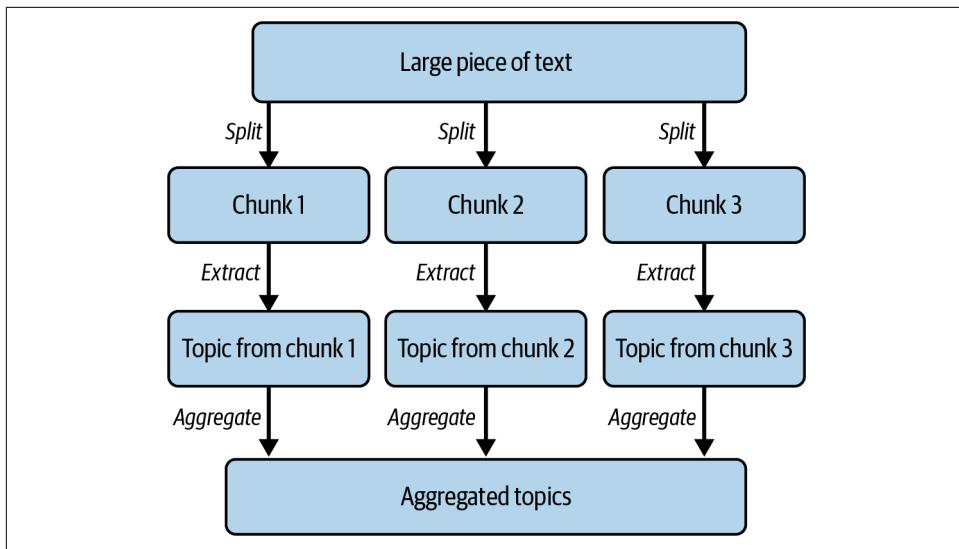


Figure 3-5. Topic extraction with an LLM after chunking text

## Benefits of Chunking Text

There are several advantages to chunking text, which include:

### *Fitting within a given context length*

LLMs only have a certain amount of input and output tokens, which is called a *context length*. By reducing the input tokens you can make sure the output won't be cut off and the initial request won't be rejected.

### *Reducing cost*

Chunking helps you to only retrieve the most important points from documents, which reduces your token usage and API costs.

### *Improved performance*

Chunking reduces the processing load on LLMs, allowing for faster response times and more efficient resource utilization.

### *Increased flexibility*

Chunking allows developers to tailor AI responses based on the specific needs of a given task or application.

## Scenarios for Chunking Text

Chunking text can be particularly beneficial in certain scenarios, while in others it may not be required. Understanding when to apply this technique can help in optimizing the performance and cost efficiency of LLMs.

## **When to chunk**

### *Large documents*

When dealing with extensive documents that exceed the maximum token limit of the LLM

### *Complex analysis*

In scenarios where a detailed analysis is required and the document needs to be broken down for better comprehension and processing

### *Multitopic documents*

When a document covers multiple topics and it's beneficial to handle them individually

## **When not to chunk**

### *Short documents*

When the document is short and well within the token limits of the LLM

### *Simple analysis*

In cases where the analysis or processing required is straightforward and doesn't benefit from chunking

### *Single-topic documents*

When a document is focused on a single topic and chunking doesn't add value to the processing

## **Poor Chunking Example**

When text is not chunked correctly, it can lead to reduced LLM performance. Consider the following paragraph from a news article:

The local council has decided to increase the budget for education by 10% this year, a move that has been welcomed by parents and teachers alike. The additional funds will be used to improve school infrastructure, hire more teachers, and provide better resources for students. However, some critics argue that the increase is not enough to address the growing demands of the education system.

When the text is fragmented into isolated words, the resulting list lacks the original context:

```
["The", "local", "council", "has", "decided", "to", "increase", "the",
 "budget", ...]
```

The main issues with this poor chunking example include:

#### *Loss of context*

By splitting the text into individual words, the original meaning and relationships between the words are lost. This makes it difficult for AI models to understand and respond effectively.

#### *Increased processing load*

Processing individual words requires more computational resources, making it less efficient than processing larger chunks of text.

As a result of the poor chunking in this example, an LLM may face several challenges:

- Difficulty understanding the main ideas or themes of the text
- Struggling to generate accurate summaries or translations
- Inability to effectively perform tasks such as sentiment analysis or text classification

By understanding the pitfalls of poor chunking, you can apply prompt engineering principles to improve the process and achieve better results with AI language models.

Let's explore an improved chunking example using the same news article paragraph from the previous section; you'll now chunk the text by sentence:

```
[ """The local council has decided to increase the budget for education  
by 10% this year, a move that has been welcomed by parents and teachers alike.  
""" ,  
    """The additional funds will be used to improve school infrastructure,  
hire more teachers, and provide better resources for students.""" ,  
    """However, some critics argue that the increase is not enough to  
address the growing demands of the education system.""" ]
```



#### **Divide Labor and Evaluate Quality**

Define the granularity at which the text should be chunked, such as by sentence, paragraph, or topic. Adjust parameters like the number of tokens or model temperature to optimize the chunking process.

By chunking the text in this manner, you could insert whole sentences into an LLM prompt with the most relevant sentences.

# Chunking Strategies

There are many different chunking strategies, including:

## *Splitting by sentence*

Preserves the context and structure of the original content, making it easier for LLMs to understand and process the information. Sentence-based chunking is particularly useful for tasks like summarization, translation, and sentiment analysis.

## *Splitting by paragraph*

This approach is especially effective when dealing with longer content, as it allows the LLM to focus on one cohesive unit at a time. Paragraph-based chunking is ideal for applications like document analysis, topic modeling, and information extraction.

## *Splitting by topic or section*

This method can help AI models better identify and understand the main themes and ideas within the content. Topic-based chunking is well suited for tasks like text classification, content recommendations, and clustering.

## *Splitting by complexity*

For certain applications, it might be helpful to split text based on its complexity, such as the reading level or technicality of the content. By grouping similar complexity levels together, LLMs can more effectively process and analyze the text. This approach is useful for tasks like readability analysis, content adaptation, and personalized learning.

## *Splitting by length*

This technique is particularly helpful when working with very long or complex documents, as it allows LLMs to process the content more efficiently. Length-based chunking is suitable for applications like large-scale text analysis, search engine indexing, and text preprocessing.

## *Splitting by tokens using a tokenizer*

Utilizing a tokenizer is a crucial step in many natural language processing tasks, as it enables the process of splitting text into individual tokens. Tokenizers divide text into smaller units, such as words, phrases, or symbols, which can then be analyzed and processed by AI models more effectively. You'll shortly be using a package called `tiktoken`, which is a bytes-pair encoding tokenizer (BPE) for chunking.

**Table 3-1** provides a high-level overview of the different chunking strategies; it's worth considering what matters to you most when performing chunking.

Are you more interested in preserving semantic context, or would naively splitting by length suffice?

*Table 3-1. Six chunking strategies highlighting their advantages and disadvantages*

| Splitting strategy                     | Advantages   | Disadvantages  |
|--|--|--|
| Splitting by sentence                  | Preserves context, suitable for various tasks                          | May not be efficient for very long content                   |
| Splitting by paragraph                 | Handles longer content, focuses on cohesive units                      | Less granularity, may miss subtle connections                |
| Splitting by topic                     | Identifies main themes, better for classification                      | Requires topic identification, may miss fine details         |
| Splitting by complexity                | Groups similar complexity levels, adaptive                             | Requires complexity measurement, not suitable for all tasks  |
| Splitting by length                    | Manages very long content, efficient processing                        | Loss of context, may require more preprocessing steps        |
| Using a tokenizer: Splitting by tokens | Accurate token counts, which helps in avoiding LLM prompt token limits | Requires tokenization, may increase computational complexity |

By choosing the appropriate chunking strategy for your specific use case, you can optimize the performance and accuracy of AI language models.

## Sentence Detection Using SpaCy

*Sentence detection*, also known as sentence boundary disambiguation, is the process used in NLP that involves identifying the start and end of sentences within a given text. It can be particularly useful for tasks that require preserving the context and structure of the original content. By splitting the text into sentences, LLMs can better understand and process the information for tasks such as summarization, translation, and sentiment analysis.

Splitting by sentence is possible using NLP libraries such as spaCy. Ensure that you have spaCy installed in your Python environment. You can install it with `pip install spacy`. Download the `en_core_web_sm` model using the command `python -m spacy download en_core_web_sm`.

In [Example 3-3](#), the code demonstrates sentence detection using the spaCy library in Python.

*Example 3-3. Sentence detection with spaCy*

```
import spacy  
  
nlp = spacy.load("en_core_web_sm")
```

```
text = "This is a sentence. This is another sentence."  
  
doc = nlp(text)  
  
for sent in doc.sents:  
    print(sent.text)
```

Output:

```
This is a sentence.  
This is another sentence.
```

First, you'll import the spaCy library and load the English model (`en_core_web_sm`) to initialize an `nlp` object. Define an input text with two sentences; the text is then processed with `doc = nlp(text)`, creating a `doc` object as a result. Finally, the code iterates through the detected sentences using the `doc.sents` attribute and prints each sentence.

## Building a Simple Chunking Algorithm in Python

After exploring many chunking strategies, it's important to build your intuition by writing a simple chunking algorithm from scratch.

**Example 3-4** shows how to chunk text based on the length of characters from the blog post “Hubspot - What Is Digital Marketing?” This file can be found in the Github repository at [content/chapter\\_3/hubspot\\_blog\\_post.txt](#).

To correctly read the `hubspot_blog_post.txt` file, make sure your current working directory is set to the [content/chapter\\_3](#) GitHub directory. This applies for both running the Python code or launching the Jupyter Notebook server.

*Example 3-4. Character chunking*

```
with open("hubspot_blog_post.txt", "r") as f:  
    text = f.read()  
  
chunks = [text[i : i + 200] for i in range(0, len(text), 200)]  
  
for chunk in chunks:  
    print("-" * 20)  
    print(chunk)
```

Output:

```
search engine optimization strategy for many local businesses is an optimized  
Google My Business profile to appear in local search results when people look for  
products or services related to what yo  
-----  
u offer.
```

For Keeps Bookstore, a local bookstore in Atlanta, GA, has optimized its Google My Business profile for local SEO so it appears in queries for “atlanta bookstore.”

-----  
...(shortened for brevity)...

First, you open the text file `hubspot_blog_post.txt` with the `open` function and read its contents into the variable `text`. Then using a list comprehension you create a list of chunks, where each chunk is a 200 character substring of text.

Then you use the `range` function to generate indices for each 200 character substring, and the `i:i+200` slice notation to extract the substring from `text`.

Finally, you loop through each chunk in the `chunks` list and `print` it to the console.

As you can see, because the chunking implementation is relatively simple and only based on length, there are gaps within the sentences and even words.

For these reasons we believe that good NLP chunking has the following properties:

- Preserves entire words, ideally sentences and contextual points made by speakers
- Handles for when sentences span across several pages, for example, page 1 into page 2
- Provides an adequate token count for each chunk so that the total number of input tokens will appropriately fit into a given token context window for any LLM

## Sliding Window Chunking

*Sliding window chunking* is a technique used for dividing text data into overlapping chunks, or *windows*, based on a specified number of characters, tokens, or words.

But what exactly is a sliding window?

Imagine viewing a long piece of text through a small window. This window is only capable of displaying a fixed number of items at a time. As you slide this window from the beginning to the end of the text, you see *overlapping chunks of text*. This mechanism forms the essence of the sliding window approach.

Each window size is defined by a *fixed number of characters, tokens, or words*, and the *step size* determines how far the window moves with each slide.

In [Figure 3-6](#), with a window size of 4 words and a step size of 1, the first chunk would contain the first 4 words of the text. The window then slides 1 word to the right to create the second chunk, which contains words 2 through 5.

This process repeats until the end of the text is reached, ensuring each chunk overlaps with the previous and next ones to retain some shared context.

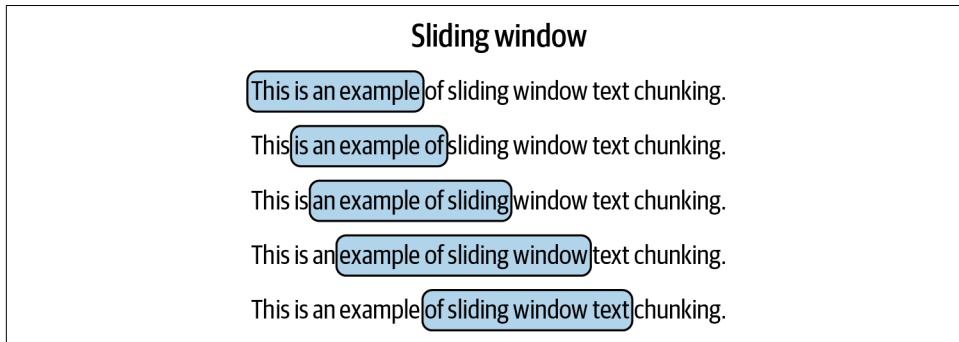


Figure 3-6. A sliding window, with a window size of 4 and a step size of 1

Due to the step size being 1, there is a lot of duplicate information between chunks, and at the same time the risk of losing information between chunks is dramatically reduced.

This is in stark contrast to [Figure 3-7](#), which has a window size of 4 words and a step size of 2. You'll notice that because of the 100% increase in step size, the amount of information shared between the chunks is greatly reduced.

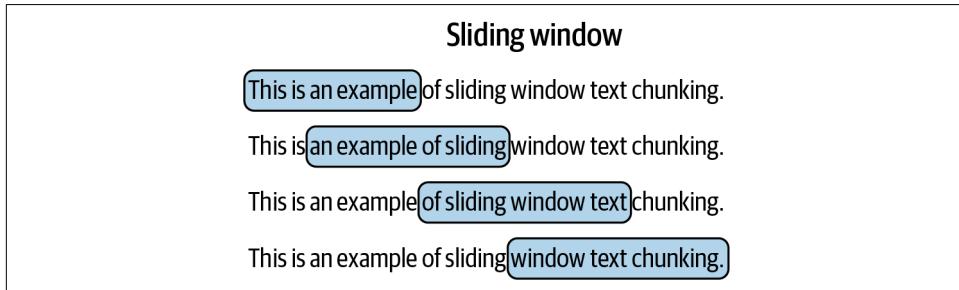


Figure 3-7. A sliding window, with a window size of 4 and a step size of 2

You will likely need a larger overlap if accuracy and preserving semantic context are more important than minimizing token inputs or the number of requests made to an LLM.

[Example 3-5](#) shows how you can implement a sliding window using Python's `len()` function. The `len()` function provides us with the total number of characters rather than words in a given text string, which subsequently aids in defining the parameters of our sliding windows.

### *Example 3-5. Sliding window*

```
def sliding_window(text, window_size, step_size):
    if window_size > len(text) or step_size < 1:
        return []
    return [text[i:i+window_size] for i
            in range(0, len(text) - window_size + 1, step_size)]

text = "This is an example of sliding window text chunking."
window_size = 20
step_size = 5

chunks = sliding_window(text, window_size, step_size)

for idx, chunk in enumerate(chunks):
    print(f"Chunk {idx + 1}: {chunk}")
```

This code outputs:

```
Chunk 1: This is an example o
Chunk 2: is an example of sli
Chunk 3: example of sliding
Chunk 4: ple of sliding windo
Chunk 5: f sliding window tex
Chunk 6: ding window text chu
Chunk 7: window text chunking
```

In the context of prompt engineering, the sliding window approach offers several benefits over fixed chunking methods. It allows LLMs to retain a higher degree of context, as there is an overlap between the chunks and offers an alternative approach to preserving context compared to sentence detection.

## Text Chunking Packages

When working with LLMs such as GPT-4, always remain wary of the maximum context length:

```
maximum_context_length = input_tokens + output_tokens
```

There are various tokenizers available to break your text down into manageable units, the most popular ones being NLTK, spaCy, and tiktoken.

Both **NLTK** and **spaCy** provide comprehensive support for text processing, but you'll be focusing on tiktoken.

# Text Chunking with Tiktoker

**Tiktoker** is a fast *byte pair encoding (BPE)* tokenizer that breaks down text into subword units and is designed for use with OpenAI's models. Tiktoker offers faster performance than comparable open source tokenizers.

As a developer working with GPT-4 applications, using tiktoker offers you several key advantages:

## *Accurate token breakdown*

It's crucial to divide text into tokens because GPT models interpret text as individual tokens. Identifying the number of tokens in your text helps you figure out whether the text is too lengthy for a model to process.

## *Effective resource utilization*

Having the correct token count enables you to manage resources efficiently, particularly when using the OpenAI API. Being aware of the exact number of tokens helps you regulate and optimize API usage, maintaining a balance between costs and resource usage.

# Encodings

Encodings define the method of converting text into tokens, with different models utilizing different encodings. Tiktoker supports three encodings commonly used by OpenAI models:

| Encoding name       | OpenAI models                                    |
|---------------------|--|
| c100k_base          | GPT-4, GPT-3.5-turbo, text-embedding-ada-002     |
| p50k_base           | Codex models, text-davinci-002, text-davinci-003 |
| r50k_base (or gpt2) | GPT-3 models like davinci                        |

# Understanding the Tokenization of Strings

In English, tokens can vary in length, ranging from a single character like *t*, to an entire word such as *great*. This is due to the adaptable nature of tokenization, which can accommodate even tokens shorter than a character in complex script languages or tokens longer than a word in languages without spaces or where phrases function as single units.

It is not uncommon for spaces to be included within tokens, such as "is" rather than "is " or "+is". This practice helps maintain the original text formatting and can capture specific linguistic characteristics.



To easily examine the tokenization of a string, you can use [OpenAI Tokenizer](#).

You can install `tiktoken` from PyPI with `pip install tiktoken`. In the following example, you'll see how to easily encode text into tokens and decode tokens into text:

```
# 1. Import the package:  
import tiktoken  
  
# 2. Load an encoding with tiktoken.get_encoding()  
encoding = tiktoken.get_encoding("cl100k_base")  
  
# 3. Turn some text into tokens with encoding.encode()  
# while turning tokens into text with encoding.decode()  
print(encoding.encode("Learning how to use Tiktoken is fun!"))  
print(encoding.decode([1061, 15009, 374, 264, 2294, 1648,  
311, 4048, 922, 15592, 0]))  
  
# [48567, 1268, 311, 1005, 73842, 5963, 374, 2523, 0]  
# "Data engineering is a great way to learn about AI!"
```

Additionally let's write a function that will tokenize the text and then count the number of tokens given a `text_string` and `encoding_name`.

```
def count_tokens(text_string: str, encoding_name: str) -> int:  
    """  
        Returns the number of tokens in a text string using a given encoding.  
  
    Args:  
        text: The text string to be tokenized.  
        encoding_name: The name of the encoding to be used for tokenization.  
  
    Returns:  
        The number of tokens in the text string.  
  
    Raises:  
        ValueError: If the encoding name is not recognized.  
    """  
    encoding = tiktoken.get_encoding(encoding_name)  
    num_tokens = len(encoding.encode(text_string))  
    return num_tokens  
  
# 4. Use the function to count the number of tokens in a text string.  
text_string = "Hello world! This is a test."  
print(count_tokens(text_string, "cl100k_base"))
```

This code outputs 8.

# Estimating Token Usage for Chat API Calls

ChatGPT models, such as GPT-3.5-turbo and GPT-4, utilize tokens similarly to previous completion models. However, the message-based structure makes token counting for conversations more challenging:

```
def num_tokens_from_messages(messages, model="gpt-3.5-turbo-0613"):
    """Return the number of tokens used by a list of messages."""
    try:
        encoding = tiktoken.encoding_for_model(model)
    except KeyError:
        print("Warning: model not found. Using cl100k_base encoding.")
        encoding = tiktoken.get_encoding("cl100k_base")
    if model in {
        "gpt-3.5-turbo-0613",
        "gpt-3.5-turbo-16k-0613",
        "gpt-4-0314",
        "gpt-4-32k-0314",
        "gpt-4-0613",
        "gpt-4-32k-0613",
    }:
        tokens_per_message = 3
        tokens_per_name = 1
    elif model == "gpt-3.5-turbo-0301":
        tokens_per_message = 4 # every message follows
        # </start/>{role/name}\n{content}</end/>\n
        tokens_per_name = -1 # if there's a name, the role is omitted
    elif "gpt-3.5-turbo" in model:
        print('Warning: gpt-3.5-turbo may update over time. Returning
              num tokens assuming gpt-3.5-turbo-0613.')
        return num_tokens_from_messages(messages, model="gpt-3.5-turbo-0613")
    elif "gpt-4" in model:
        print('Warning: gpt-4 may update over time.
              Returning num tokens assuming gpt-4-0613.')
        return num_tokens_from_messages(messages, model="gpt-4-0613")
    else:
        raise NotImplementedError(
            f"""num_tokens_from_messages() is not implemented for model
            {model}."""
        )
    num_tokens = 0
    for message in messages:
        num_tokens += tokens_per_message
        for key, value in message.items():
            num_tokens += len(encoding.encode(value))
            if key == "name":
                num_tokens += tokens_per_name
    num_tokens += 3 # every reply is primed with
    # </start/>assistant</message/>
    return num_tokens
```

**Example 3-6** highlights the specific structure required to make a request against any of the chat models, which are currently GPT-3x and GPT-4.

Normally, chat history is structured with a `system` message first, and then succeeded by alternating exchanges between the `user` and the `assistant`.

*Example 3-6. A payload for the Chat Completions API on OpenAI*

```
example_messages = [
    {
        "role": "system",
        "content": '''You are a helpful, pattern-following assistant that
translates corporate jargon into plain English.'''
    },
    {
        "role": "system",
        "name": "example_user",
        "content": "New synergies will help drive top-line growth.",
    },
    {
        "role": "system",
        "name": "example_assistant",
        "content": "Things working well together will increase revenue.",
    },
    {
        "role": "system",
        "name": "example_user",
        "content": '''Let's circle back when we have more bandwidth to touch
base on opportunities for increased leverage.'''
    },
    {
        "role": "system",
        "name": "example_assistant",
        "content": "Let's talk later when we're less busy about how to
do better.",
    },
    {
        "role": "user",
        "content": '''This late pivot means we don't have
time to boil the ocean for the client deliverable.'''
    },
]
for model in ["gpt-3.5-turbo-0301", "gpt-4-0314"]:
    print(model)
    # example token count from the function defined above
    print(f'''{num_tokens_from_messages(example_messages, model)}
prompt tokens counted by num_tokens_from_messages().'''')
```

**"role": "system"**: "system" describes a system message that's useful for *providing prompt instructions*. It offers a means to tweak the assistant's character or provide explicit directives regarding its interactive approach. It's crucial to understand, though, that the system command isn't a prerequisite, and the model's default demeanor without a system command could closely resemble the behavior of "You are a helpful assistant."

The roles that you can have are ["system", "user", "assistant"].

**"content": "Some content"**: "Some content" is where you place the prompt or responses from a language model, depending upon the message's role. It can be either "assistant", "system", or "user".

## Sentiment Analysis

*Sentiment analysis* is a widely used NLP technique that helps in identifying, extracting, and understanding the emotions, opinions, or sentiments expressed in a piece of text. By leveraging the power of LLMs like GPT-4, sentiment analysis has become an essential tool for businesses, researchers, and developers across various industries.

The primary goal of sentiment analysis is to determine the attitude or emotional tone conveyed in a text, whether it's positive, negative, or neutral. This information can provide valuable insights into consumer opinions about products or services, help monitor brand reputation, and even assist in predicting market trends.

The following are several prompt engineering techniques for creating effective sentiment analysis prompts:

Input:

Is this text positive or negative?

I absolutely love the design of this phone, but the battery life is quite disappointing.

Output:

The text has a mixed tone, as it contains both positive and negative aspects. The positive part is "I absolutely love the design of this phone," while the negative part is "the battery life is quite disappointing."

Although GPT-4 identifies a "mixed tone," the outcome is a result of several shortcomings in the prompt:

*Lack of clarity*

The prompt does not clearly define the desired output format.

### *Insufficient examples*

The prompt does not include any examples of positive, negative, or neutral sentiments, which could help guide the LLM in understanding the distinctions between them.

### *No guidance on handling mixed sentiments*

The prompt does not specify how to handle cases where the text contains a mix of positive and negative sentiments.

Input:

Using the following examples as a guide:

positive: 'I absolutely love the design of this phone!'

negative: 'The battery life is quite disappointing.'

neutral: 'I liked the product, but it has short battery life.'

Only return either a single word of:

- positive
- negative
- neutral

Please classify the sentiment of the following text as positive, negative, or neutral: I absolutely love the design of this phone, but the battery life is quite disappointing.

Output:

neutral

This prompt is much better because it:

### *Provides clear instructions*

The prompt clearly states the task, which is to classify the sentiment of the given text into one of three categories: positive, negative, or neutral.

### *Offers examples*

The prompt provides examples for each of the sentiment categories, which helps in understanding the context and desired output.

### *Defines the output format*

The prompt specifies that the output should be a single word, ensuring that the response is concise and easy to understand.

## Techniques for Improving Sentiment Analysis

To enhance sentiment analysis accuracy, preprocessing the input text is a vital step. This involves the following:

### *Special characters removal*

Exceptional characters such as emojis, hashtags, and punctuation may skew the rule-based sentiment algorithm's judgment. Besides, these characters might not be recognized by machine learning and deep learning models, resulting in misclassification.

### *Lowercase conversion*

Converting all the characters to lowercase aids in creating uniformity. For instance, words like *Happy* and *happy* are treated as different words by models, which can cause duplication and inaccuracies.

### *Spelling correction*

Spelling errors can cause misinterpretation and misclassification. Creating a spell-check pipeline can significantly reduce such errors and improve results.

For industry- or domain-specific text, embedding domain-specific content in the prompt helps in navigating the LLM's sense of the text's framework and sentiment. It enhances accuracy in the classification and provides a heightened understanding of particular jargon and expressions.

## Limitations and Challenges in Sentiment Analysis

Despite the advancements in LLMs and the application of prompt engineering techniques, sentiment analysis still faces some limitations and challenges:

### *Handling sarcasm and irony*

Detecting sarcasm and irony in text can be difficult for LLMs, as it often requires understanding the context and subtle cues that humans can easily recognize. Misinterpreting sarcastic or ironic statements may lead to inaccurate sentiment classification.

### *Identifying context-specific sentiment*

Sentiment analysis can be challenging when dealing with context-specific sentiments, such as those related to domain-specific jargon or cultural expressions. LLMs may struggle to accurately classify sentiments in these cases without proper guidance or domain-specific examples.

## Least to Most

The *least to most* technique in prompt engineering is a powerful method for sequentially generating or extracting increasingly detailed knowledge on a given topic. This method is particularly effective when dealing with complex subjects or when a high level of detail is necessary.

Least to most uses a *chain* of prompts where each new prompt is based on the last answer. This step-by-step approach helps gather more detailed information each time, making it easier to dive deeper into any topic.

This technique can also be applied to code generation, as demonstrated in a Flask Hello World app example.

## Planning the Architecture

Before diving into the architecture, let's briefly understand what Flask is. **Flask** is a lightweight web application framework in Python, widely used for creating web applications quickly and with minimal code. (Flask is only used for demonstration purposes here and isn't included within the *requirements.txt* file for the book.)

Now, let's ask an LLM to outline the basic architecture for a simple Flask "Hello World" application.

Input:

```
Outline the basic architecture for a simple Flask "Hello World" application.
```

Output:

```
Sure! Here's an outline for a basic Flask "Hello World" application:
```

```
Import Flask module: Import the Flask module  
to use its (((("Flask module code generation"))))functionality.
```

```
Create an instance of the Flask application: Create an instance of the  
Flask class to represent your web application.
```

```
Define a route and view function: ...
```

```
Implement the view function: ...
```

```
Run the application: Start the Flask development server to run the application.
```

## Coding Individual Functions

Before coding, let's clarify what a Flask route is: it's a function linked to a URL pattern that determines what users see on a specific web page. Next, we'll provide the code snippet for a simple "Hello World" route in Flask.

Input:

Provide a code snippet for a simple "Hello World" route in Flask.

Output:

Certainly! Here's a simple code snippet for a "Hello World" route using Flask:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run()
```

In this code, we import the Flask module and create a Flask application instance named `app`. We then define a route using the `@app.route` decorator, specifying the root URL path `'/'`. The `hello_world` function is associated with this route, and it returns the string `'Hello, World!'` as the response.

The Flask application will start by calling `app.run()` when your Python script is executed directly.

## Adding Tests

Finally, let's ask it to generate a test case for the "Hello World" route in the Flask application.

Input:

Provide an example of a test case for the "Hello World" route in a Flask application.

Output:

Sure! Here's an example of a test case for the "Hello World" route in a Flask application:

```
import unittest
from flask import Flask
from flask.testing import FlaskClient

# Create the Flask application
app = Flask(__name__)

# Create the route
@app.route('/')
def hello_world():
    return 'Hello, World!'
```

```

class FlaskAppTestCase(unittest.TestCase):
    def setUp(self):
        # Create a test client
        self.app = app.test_client()

    def test_hello_world_route(self):
        # Send a GET request to the route
        response = self.app.get('/')

        # Check the response status code
        self.assertEqual(response.status_code, 200)

        # Check the response data
        self.assertEqual(response.data, b'Hello, World!')

    if __name__ == '__main__':
        unittest.main()

```

By using the least to most prompt technique, you can gradually build up the knowledge and understanding required for the task at hand.

## Benefits of the Least to Most Technique

This method is particularly useful for complex tasks, as it allows an LLM to generate relevant knowledge that will subsequently be used as context for future tasks.

Let's dive deeper into the benefits of using this approach in various other scenarios:

### *Progressive exploration*

Breaking a complex problem into smaller tasks allows an LLM to provide more detailed and accurate information at each step. This approach is especially helpful when working with a new subject matter or a multifaceted problem.

### *Flexibility*

The least to most technique offers flexibility in addressing different aspects of a problem. It enables you to pivot, explore alternative solutions, or dive deeper into specific areas as needed.

### *Improved comprehension*

By breaking down a task into smaller steps, an LLM can deliver information in a more digestible format, making it easier for you to understand and follow.

### *Collaborative learning*

This technique promotes collaboration between you and an LLM, as it encourages an iterative process of refining the output and adjusting your responses to achieve the desired outcome.

## Challenges with the Least to Most Technique

### *Overreliance on previously generated knowledge*

Using previous chat history to store the state may lead to larger tasks forgetting their initial tasks/outputs due to limitations in context length.

### *Dependence on prior prompts*

Since each prompt is built upon preceding outputs, it is imperative to ensure that the quality and responses of previous prompts provide ample information for the next step.



### Evaluate Quality

In the process of designing prompts, make sure to evaluate the quality of each prior LLM response. The performance of the next task depends upon the quality of information provided by the preceding one.

## Role Prompting

*Role prompting* is a technique in which the AI is given a specific role or character to assume while generating a response. This helps guide the AI's response style and content to better align with the user's needs or desired outcome.



[Awesome ChatGPT prompts](#) provides an extensive list of role prompts that you can use.

Let's ask ChatGPT to take on the role of a tech reviewer for MongoDB.

Input:

I want you to act as a tech reviewer. I will give you the name of a piece of technology and you will provide me with an in-depth review - including pros, cons, features, and comparisons to other technologies on the market. My first suggestion request is: MongoDB.

Output:

MongoDB is a popular NoSQL, document-oriented database management system designed to handle high-volume data storage and retrieval.

Features:

1. Document-Oriented Storage: MongoDB stores data in a flexible, JSON-like format called BSON, making it easy to store complex data structures like arrays and nested objects.

...

**Pros:**

1. High Performance: MongoDB's indexing and caching capabilities deliver fast read and write operations.

...

**Cons:**

1. Limited Support for Transactions: While MongoDB supports multidocument ACID transactions, it may not be the best choice for applications that require complex transactional operations.

...

**Comparisons to other technologies:**

1. Couchbase, another popular NoSQL database, offers similar features to MongoDB, including flexible schema and JSON-based storage. However, Couchbase is more focused on distributed caching and may be more suitable for applications requiring low-latency data access.

**Conclusion:**

MongoDB is a powerful and versatile NoSQL database solution suitable for various applications, particularly those requiring high performance and scalability. Its flexible data model, rich query language, and strong consistency make it a popular choice among developers.

The prompt is an excellent example of role prompting, as it clearly defines the role the AI should assume (a tech reviewer) and sets expectations for the type of response desired (an in-depth review of MongoDB).



### **Give Direction**

When crafting prompts, consider assigning a specific role to the AI. This sets the proper context for the response, creating a more focused and relevant output.

## **Benefits of Role Prompting**

Role prompting helps narrow down the AI's responses, ensuring more focused, contextually appropriate, and tailored results. It can also enhance creativity by pushing the AI to think and respond from unique perspectives.

## **Challenges of Role Prompting**

Role prompting can pose certain challenges. There might be potential risks for bias or stereotyping based on the role assigned. Assigning stereotyped roles can lead to generating biased responses, which could harm usability or offend individuals.

Additionally, maintaining consistency in the role throughout an extended interaction can be difficult. The model might drift off-topic or respond with information irrelevant to the assigned role.



### Evaluate Quality

Consistently check the quality of the LLM's responses, especially when role prompting is in play. Monitor if the AI is sticking to the role assigned or if it is veering off-topic.

## When to Use Role Prompting

Role prompting is particularly useful when you want to:

#### *Elicit specific expertise*

If you need a response that requires domain knowledge or specialized expertise, role prompting can help guide the LLM to generate more informed and accurate responses.

#### *Tailor response style*

Assigning a role can help an LLM generate responses that match a specific tone, style, or perspective, such as a formal, casual, or humorous response.

#### *Encourage creative responses*

Role prompting can be used to create fictional scenarios or generate imaginative answers by assigning roles like a storyteller, a character from a novel, or a historical figure.

- *Explore diverse perspectives:* If you want to explore different viewpoints on a topic, role prompting can help by asking the AI to assume various roles or personas, allowing for a more comprehensive understanding of the subject.
- *Enhance user engagement:* Role prompting can make interactions more engaging and entertaining by enabling an LLM to take on characters or personas that resonate with the user.

If you're using OpenAI, then the best place to add a role is within the System Message for chat models.

## GPT Prompting Tactics

So far you've already covered several prompting tactics, including asking for context, text style bundling, least to most, and role prompting.

Let's cover several more tactics, from managing potential hallucinations with appropriate reference text, to providing an LLM with critical *thinking time*, to understanding the concept of *task decomposition*—we have plenty for you to explore.

These methodologies have been designed to significantly boost the precision of your AI's output and are recommended by [OpenAI](#). Also, each tactic utilizes one or more of the prompt engineering principles discussed in [Chapter 1](#).

## Avoiding Hallucinations with Reference

The first method for avoiding text-based hallucinations is to instruct the model to *only answer using reference text*.

By supplying an AI model with accurate and relevant information about a given query, the model can be directed to use this information to generate its response.

Input:

```
Refer to the articles enclosed within triple quotes to respond to queries.
```

```
You must follow the following principles:
```

```
- In cases where the answer isn't found within these articles, simply return "I could not find an answer".
```

```
"""
```

```
B2B clients tend to have longer decision-making processes, and thus longer sales funnels. Relationship-building strategies work better for these clients, whereas B2C customers tend to respond better to short-term offers and messages.
```

```
"""
```

```
Example responses:
```

```
- I could not find an answer.  
- Yes, B2B clients tend to have longer decision-making processes, and thus longer sales funnels.
```

Output:

```
Yes, B2B clients tend to have longer decision-making processes, which leads to longer sales cycles.
```

If you were to ask the same reference text this question:

Input:

```
...The rest of the prompt...
```

```
Question: Are B2C sales more cost-effective?
```

Output:

```
I could not find an answer.
```



## Give Direction and Specify Format

The preceding prompt is excellent as it both instructs the model on how to find answers and also sets a specific response format for any unanswerable questions.

Considering the constrained context windows of GPTs, a method for dynamically retrieving information relevant to the asked query might be necessary to utilize this strategy.

Another approach is to direct the model to *incorporate references* from a given text in its response. When the input is enhanced with relevant information, the model can be guided to include citations in its responses by referring to sections of the supplied documents. This approach has the added benefit that citations in the output can be *authenticated automatically by matching strings* within the given documents.

Input:

You will be provided with a document delimited by triple quotes and a question. Your task is to answer the question using only the provided document and to cite the passage(s) of the document used to answer the question. If the document does not contain the information needed to answer this question then simply write: "Insufficient information." If an answer to the question is provided, it must be annotated with a citation. Use the following format to cite relevant passages ({"citation": ...}).

"""

A nuclear reactor is a device that initiates, controls, and maintains a nuclear chain reaction. The nuclear reactor uses the energy released by nuclear fission to generate heat, which is then often used to produce steam to drive a turbine and produce electricity.

-Excerpted from [https://en.wikipedia.org/wiki/Nuclear\\_reactor](https://en.wikipedia.org/wiki/Nuclear_reactor)

"""

Question: What is a nuclear reactor?

Output:

A nuclear reactor is a device that initiates, controls, and maintains a nuclear chain reaction. It uses the energy released by nuclear fission to generate heat, which is then often used to produce steam to drive a turbine and produce electricity. {"citation": "A nuclear reactor is a device that initiates, controls, and maintains a nuclear chain reaction. The nuclear reactor uses the energy released by nuclear fission to generate heat, which is then often used to produce steam to drive a turbine and produce electricity."}

## Give GPTs “Thinking Time”

Often, by explicitly guiding an LLM to *derive solutions from first principles* before reaching a verdict, you can garner more accurate responses. Providing an LLM with *thinking time* can often lead to better results.

Input:

First, try to solve the problem on your own. Then, see if your answer matches the proposed solution. Don't judge the given answer until you've tried solving it yourself.

Problem:

Jane has 5 apples, then John takes 2 from Mary. Mary originally had 15 apples. John then gives his apples to Jane.

How many apples does Jane have?

Proposed solution:

1. Jane originally had 5 apples.
2. John takes 2 apples from Mary, but this doesn't affect Jane's count.
3. John then gives his apples to Jane.

So, Jane's final count of apples is her original 5 apples plus the 2 apples that John gave her, for a total of 7 apples.

Output:

Let's break down the problem:

1. Jane originally has 5 apples.
2. John takes 2 apples from Mary. However, this action doesn't directly impact the number of apples Jane has at this point.
3. John gives his apples to Jane.

Following these steps, we need to add the 2 apples John gave to Jane to Jane's original number of apples. Therefore:

5 apples (Jane's original number) + 2 apples (given by John) = 7 apples

So, Jane now has 7 apples. My solution matches the proposed solution.

In some cases, when using GPT-4 or other generative AI models, you may not want the model's reasoning process to be visible to the user.

You can achieve this by asking an LLM to *generate an inner monologue*. This is particularly useful in tutoring applications, where revealing the model's reasoning might give away the solution prematurely.

# The Inner Monologue Tactic

The *inner monologue tactic* instructs the model to structure parts of the output that should be hidden from the user in a specific format. This makes it easy to remove these parts before presenting the final output to the user.

Here's how you can utilize this tactic to answer user queries effectively.

Input:

## Step 1: Solve the Problem

First, you need to work out your solution to the problem. Be sure not to rely on the user's solution, as it might be incorrect. You can encapsulate your workings within triple quotes ('''') to indicate that this section should be hidden.

## Step 2: Compare Solutions

Next, compare your solution with the user's solution. Evaluate whether the user's solution is correct or not. Again, encapsulate this section within triple quotes ('''').

## Step 3: Prepare a Hint

If the user made a mistake, determine an appropriate hint that can help guide them toward the correct answer, without giving it away outright. This section should also be within triple quotes ('''').

## Step 4: Deliver the Hint

If a mistake was made, present the hint from the previous step to the user. Remember not to use triple quotes for this step. Instead of using a title like "Step 4", use a heading such as `Hint:`

**Problem Statement:** A deck of cards has 52 cards: 13 hearts, 13 diamonds, 13 clubs, and 13 spades. If two cards are drawn from the deck without replacement, what is the probability that one of the drawn cards is a heart and the other one is a club?

User's Solution:  $(13/52)*(13/51) = 169/2652 = 13/204$

Output:

'''

## Step 1 - My Solution:

First, calculate the probability of drawing 1 heart and 1 club from a deck of 52 cards without replacement.

In the first draw, the probability of drawing a heart is 13/52.

```
For the second draw, after removing 1 card (assuming it's a heart), the  
total number of cards becomes 51. So, the probability of drawing a club now  
is 13/51.
```

```
Multiplying these two probabilities gives the probability of both events  
happening in sequence, which is P(Heart then Club) = (13/52)*(13/51).
```

```
Repeat the calculation assuming the first drawn card is a club and the  
second drawn card is a heart. Remember, order matters in this case as  
we're dealing with separate events.  
"""  
"""
```

```
Step 2 - Comparison with the User's Solution:
```

```
Compare your solution with the user's solution. The user's solution only  
considers the sequence of drawing a heart first and then a club. They  
have not accounted for the alternate sequence (Club then Heart), and  
thus, the solution is incomplete.  
"""  
"""
```

```
Step 3 - Preparation of Hint:
```

```
Given that the user hasn't considered both sequences (Heart then Club  
and Club then Heart), a hint at the consideration of both possible  
sequences will be helpful. This could nudge the user in the right  
direction without revealing the solution outright.  
"""
```

```
# Step 4 - Delivery of Hint:
```

```
# Hint: Remember that the two cards could be drawn in any order. Consider  
both possible sequences: Heart then Club and Club then Heart.
```

## Self-Eval LLM Responses

Another tactic you can use is to *critique a generated LLM output* and ask whether the LLM missed any information or important facts. You're essentially asking an LLM to *evaluate itself* based on its previous output.

First, create a prompt to generate a "Hello World" function.

Input:

```
Generate a "Hello World" Python function.
```

Output:

```
Sure, here's a very simple Python function that prints "Hello, World!" to  
the console:
```

```
def print_hello_world():  
    print("Hello, World!")  
  
print_hello_world()
```

When you run this function, it will print the string "Hello, World!" to the console.

The LLM returns a simple Python function called `print_hello_world()` that prints the traditional "Hello, World!" greeting.

Then make a second LLM request with the previous chat history, asking for the initial output to be improved.

Input:

```
...Previous Chat History...
```

Given the previous function, can you improve it again with anything else?  
Perhaps defaults or type hints?

Output:

We can add type hints to make it clear that the function doesn't take any arguments and doesn't return anything. We can also add a docstring to describe what the function does.

Here's the improved code:

```
def print_hello_world() -> None:  
    """  
    Prints "Hello, World!" to the standard output.  
    """  
    print("Hello, World!")
```

If you wanted to make the function more flexible, you could add a default argument that allows you to print a different message:

```
def print_message(message: str = "Hello, World!") -> None:  
    """  
    Prints the given message to the standard output. If no message is given  
    , it defaults to "Hello, World!".  
  
    Args:  
        message (str, optional): The message to be printed.  
        Defaults to "Hello, World!".  
    """  
    print(message)
```

Consequently GPT-4 returns an upgraded version of the function, now furnished with Python type hints and a default argument. This enriches the function with greater flexibility, allowing it to print not just "Hello, World!" but any user-specified message.

These prompt-response exchanges illustrate how you can easily refine generated LLM outputs until you're satisfied with the final output.



It's possible to critique an LLM's response multiple times, until no further refinement is provided by the LLM.

## Classification with LLMs

Classifying, in the context of AI, refers to the process of predicting the class or category of a given data point or sample. It's a common task in machine learning where models are trained to assign predefined labels to unlabeled data based on learned patterns.

LLMs are powerful assets when it comes to classification, even with zero or only a small number of examples provided within a prompt. Why? That's because LLMs, like GPT-4, have been previously trained on an extensive dataset and now possess a degree of reasoning.

There are two overarching strategies in solving classification problems with LLMs: *zero-shot learning* and *few-shot learning*.

### *Zero-shot learning*

In this process, the LLM classifies data with exceptional accuracy, without the aid of any prior specific examples. It's akin to acing a project without any preparation—impressive, right?

### *Few-shot learning*

Here, you provide your LLM with a small number of examples. This strategy can significantly influence the structure of your output format and enhance the overall classification accuracy.

Why is this groundbreaking for you?

Leveraging LLMs lets you sidestep lengthy processes that traditional machine learning processes demand. Therefore, you can quickly prototype a classification model, determine a base level accuracy, and create immediate business value.



Although an LLM can perform classification, depending upon your problem and training data you might find that using a traditional machine learning process could yield better results.

# Building a Classification Model

Let's explore a few-shot learning example to determine the sentiment of text into either 'Compliment', 'Complaint', or 'Neutral'.

Given the statement, classify it as either "Compliment", "Complaint", or "Neutral":

1. "The sun is shining." - Neutral
2. "Your support team is fantastic!" - Compliment
3. "I had a terrible experience with your software." - Complaint

You must follow the following principles:

- Only return the single classification word. The response should be either "Compliment", "Complaint", or "Neutral".
- Perform the classification on the text enclosed within """ delimiters.

"""The user interface is intuitive."""

**Classification:**

Compliment

Several good use cases for LLM classification include:

## *Customer reviews*

Classify user reviews into categories like "Positive," "Negative," or "Neutral." Dive deeper by further identifying subthemes such as "Usability," "Customer Support," or "Price."

## *Email filtering*

Detect the intent or purpose of emails and classify them as "Inquiry," "Complaint," "Feedback," or "Spam." This can help businesses prioritize responses and manage communications efficiently.

## *Social media sentiment analysis*

Monitor brand mentions and sentiment across social media platforms. Classify posts or comments as "Praise," "Critic," "Query," or "Neutral." Gain insights into public perception and adapt marketing or PR strategies accordingly.

## *News article categorization*

Given the vast amount of news generated daily, LLMs can classify articles by themes or topics such as "Politics," "Technology," "Environment," or "Entertainment."

## *Résumé screening*

For HR departments inundated with résumés, classify them based on predefined criteria like "Qualified," "Overqualified," "Underqualified," or categorize by expertise areas such as "Software Development," "Marketing," or "Sales."



Be aware that exposing emails, résumés, or sensitive data does run the risk of data being leaked into OpenAI's future models as training data.

## Majority Vote for Classification

Utilizing multiple LLM requests can help in reducing the variance of your classification labels. This process, known as *majority vote*, is somewhat like choosing the most common fruit out of a bunch. For instance, if you have 10 pieces of fruit and 6 out of them are apples, then apples are the majority. The same principle goes for choosing the majority vote in classification labels.

By soliciting several classifications and taking the *most frequent classification*, you're able to reduce the impact of potential outliers or unusual interpretations from a single model inference. However, do bear in mind that there can be significant downsides to this approach, including the increased time required and cost for multiple API calls.

Let's classify the same piece of text three times, and then take the majority vote:

```
from openai import OpenAI
import os

client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

base_template = """
Given the statement, classify it as either "Compliment", "Complaint", or
"Neutral":
1. "The sun is shining." - Neutral
2. "Your support team is fantastic!" - Compliment
3. "I had a terrible experience with your software." - Complaint

You must follow the following principles:
- Only return the single classification word. The response should be either
"Compliment", "Complaint", or "Neutral".
- Perform the classification on the text enclosed within ''' delimiters.

'''{content}'''

Classification:
"""

responses = []

for i in range(0, 3):
    response = client.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "system",
                   "content": base_template.format(content='''Outside is rainy,')}]
```

```

        but I am having a great day, I just don't understand how people
        live, I'm so sad!'''},{},)
responses.append(response.choices[0].message.content.strip())

def most_frequent_classification(responses):
    # Use a dictionary to count occurrences of each classification
    count_dict = {}
    for classification in responses:
        count_dict[classification] = count_dict.get(classification, 0) + 1

    # Return the classification with the maximum count
    return max(count_dict, key=count_dict.get)

print(most_frequent_classification(responses)) # Expected Output: Neutral

```

Calling the `most_frequent_classification(responses)` function should pinpoint 'Neutral' as the dominant sentiment. You've now learned how to use the OpenAI package for majority vote classification.

## Criteria Evaluation

In Chapter 1, a human-based evaluation system was used with a simple thumbs-up/thumbs-down rating system to identify how often a response met our expectations. Rating manually can be expensive and tedious, requiring a qualified human to judge quality or identify errors. While this work can be outsourced to low-cost raters on services such as Mechanical Turk, designing such a task in a way that gets valid results can itself be time-consuming and error prone. One increasingly common approach is to use a more sophisticated LLM to evaluate the responses of a smaller model.

The evidence is mixed on whether LLMs can act as effective evaluators, with some studies [claiming LLMs are human-level evaluators](#) and others [identifying inconsistencies in how LLMs evaluate](#). In our experience, GPT-4 is a useful evaluator with consistent results across a diverse set of tasks. In particular, GPT-4 is effective and reliable in evaluating the responses from smaller, less sophisticated models like GPT-3.5-turbo. In the example that follows, we generate concise and verbose examples of answers to a question using GPT-3.5-turbo, ready for rating with GPT-4.

Input:

```

from openai import OpenAI
import os

client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

responses = []

for i in range(10):
    # concise if even, verbose if odd
    style = "concise" if i % 2 == 0 else "verbose"

```

```

if style == "concise":
    prompt = f"""Return a {style} answer to the
    following question: What is the meaning of life?"""
else:
    prompt = f"""Return an answer to the following
    question: What is the meaning of life?"""

response = client.chat.completions.create(
    # using GPT-3.5 Turbo for this example
    model="gpt-3.5-turbo",
    messages=[{"role": "user",
               "content": prompt}])
responses.append(
    response.choices[0].message.content.strip())

system_prompt = """You are assessing the conciseness of a
response from a chatbot.
You only respond with a 1 if the response is concise,
and a 0 if it is not.
"""

ratings = []

for idx, response in enumerate(responses):
    rating = client.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "system",
                   "content": system_prompt},
                  {"role": "system",
                   "content": response}])
    ratings.append(
        rating.choices[0].message.content.strip())

for idx, rating in enumerate(ratings):
    style = "concise" if idx % 2 == 0 else "verbose"
    print(f"Style: {style}, ", f"Rating: {rating}")

```

Output:

```

Style: concise, Rating: 1
Style: verbose, Rating: 0

```

This script is a Python program that interacts with the OpenAI API to generate and evaluate responses based on their conciseness. Here's a step-by-step explanation:

1. `responses = []` creates an empty list named `responses` to store the responses generated by the OpenAI API.
2. The `for` loop runs 10 times, generating a response for each iteration.
3. Inside the loop, `style` is determined based on the current iteration number (`i`). It alternates between “concise” and “verbose” for even and odd iterations, respectively.
4. Depending on the `style`, a `prompt` string is formatted to ask, “What is the meaning of life?” in either a concise or verbose manner.
5. `response = client.chat.completions.create(...)` makes a request to the OpenAI API to generate a response based on the `prompt`. The model used here is specified as “gpt-3.5-turbo.”
6. The generated response is then stripped of any leading or trailing whitespace and added to the `responses` list.
7. `system_prompt = """You are assessing..."""` sets up a prompt used for evaluating the conciseness of the generated responses.
8. `ratings = []` initializes an empty list to store the conciseness ratings.
9. Another `for` loop iterates over each response in `responses`.
10. For each response, the script sends it along with the `system_prompt` to the OpenAI API, requesting a conciseness evaluation. This time, the model used is “gpt-4.”
11. The evaluation rating (either 1 for concise or 0 for not concise) is then stripped of whitespace and added to the `ratings` list.
12. The final `for` loop iterates over the `ratings` list. For each rating, it prints the `style` of the response (either “concise” or “verbose”) and its corresponding conciseness `rating`.

For simple ratings like conciseness, GPT-4 performs with near 100% accuracy; however, for more complex ratings, it’s important to spend some time evaluating the evaluator. For example, by setting test cases that contain an issue, as well as test cases that do not contain an issue, you can identify the accuracy of your evaluation metric. An evaluator can itself be evaluated by counting the number of false positives (when the LLM hallucinates an issue in a test case that is known not to contain an issue), as well as the number of false negatives (when the LLM misses an issue in a test case that is known to contain an issue). In our example we generated the concise and verbose examples, so we can easily check the rating accuracy, but in more complex examples you may need human evaluators to validate the ratings.



## Evaluate Quality

Using GPT-4 to evaluate the responses of less sophisticated models is an emerging standard practice, but care must be taken that the results are reliable and consistent.

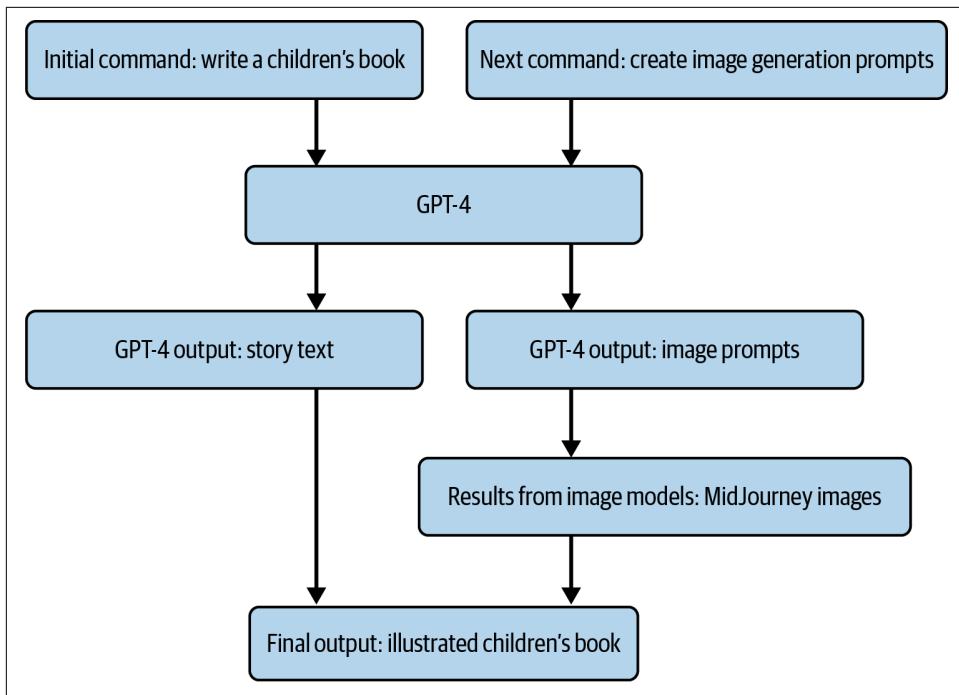
Compared to human-based evaluation, LLM-based or synthetic evaluation typically costs an order of magnitude less and completes in a few minutes rather than taking days or weeks. Even in important or sensitive cases where a final manual review by a human is necessary, rapid iteration and A/B testing of the prompt through synthetic reviews can save significant time and improve results considerably. However, the cost of running many tests at scale can add up, and the latency or rate limits of GPT-4 can be a blocker. If at all possible, a prompt engineer should first test using programmatic techniques that don't require a call to an LLM, such as simply measuring the length of the response, which runs near instantly for close to zero cost.

## Meta Prompting

*Meta prompting* is a technique that involves the creation of text prompts that, in turn, generate other text prompts. These text prompts are then used to generate new assets in many mediums such as images, videos, and more text.

To better understand meta prompting, let's take the example of authoring a children's book with the assistance of GPT-4. First, you direct the LLM to generate the text for your children's book. Afterward, you invoke meta prompting by instructing GPT-4 to produce prompts that are suitable for image-generation models. This could mean creating situational descriptions or specific scenes based on the storyline of your book, which then can be given to AI models like Midjourney or Stable Diffusion. These image-generation models can, therefore, deliver images in harmony with your AI-crafted children's story.

Figure 3-8 visually describes the process of meta prompting in the context of crafting a children's book.



*Figure 3-8. Utilizing an LLM to generate image prompts for MidJourney's image creation in the process of crafting a children's book*

Meta prompts offer a multitude of benefits for a variety of applications:

#### *Image generation from product descriptions*

Meta prompts can be employed to derive an image generation prompt for image models like [Midjourney](#), effectively creating a visual representation of product descriptions.

#### *Generating style/feature prompts*

Let's consider you are a copywriter needing to develop a unique style guide prompt from a couple of blog posts. Given each client has a distinctive tone and style, it's beneficial to utilize a *meta prompt* that encapsulates all the varied features, rather than producing a single prompt output.

#### *Optimizing prompts to achieve specific goals*

A common approach is to ask ChatGPT or another language model to refine or improve **Prompt A** in order to attain **Goal 1**, given **Prompt A** and **Goal 1**. This method aids in identifying any missed input features, that by adding could assist the language model in generating a more detailed and desirable response.

Let's experiment with creating a meta prompt for the copywriting example.

## Input:

Act as a GPT Prompt Engineer, you are solely responsible for producing highly effective large language model text prompts given a task.

You must follow the following principles:

- Return only a single prompt, never return the output of a prompt.
- You must follow the task properly.
- You must either refine the prompt or create a new prompt depending upon the Task.
- Bullet point out any important features so that another Prompt Engineer can easily copy inputs into the prompt.

Task: You must create a copywriting guide ChatGPT several blog posts.

Include relevant features within the ChatGPT prompt that will allow ChatGPT to imitate the writer. You must decide what the most important features are to mimic the writer's style etc. You will only be applying this style to new blog posts.

Blog post text: You need a variety of soft skills in addition to technical skills to succeed in the technology sector. Soft skills are used by software professionals to collaborate with their peers...

## Output:

Prompt:

Based on the extensive text provided, your task is to create a series of blog posts following the same style and structure of the writer. As a guide you should follow these features:

1. Clear, direct language: Your writing should be in clear and direct language, breaking down complex subjects into easily understandable concepts.
2. Use of subheadings: To provide structure to your blog posts, break them up with relevant subheadings. This helps to guide the reader and make your text more readable.
- ...
9. Inclusion of technical terminology: When discussing technical topics, use appropriate technical language to ensure accuracy and credibility. However, be sure to provide explanations or definitions for more complex terms.

As previously mentioned, it can be beneficial to *critique the model's output* and to ask for a refined prompt either via the interface or API.

ChatGPT can perform web searches and can make use of the content of web pages, along with the previously generated copywriting style guide. Let's provide two URLs that ChatGPT will read to enhance the output.

Input:

Given the following style guide and these two blog posts, I want you to improve the copywriting style prompt so an LLM or copywriter can write effectively for my client.

...Previously Generated Copywriting guide...

URL 1: <https://understandingdata.com/posts/how-to-clean-and-process-data>  
URL 2: <https://understandingdata.com/posts/how-to-scrape-twitter-data>

Figure 3-9 shows the output.

 ChatGPT

I've now browsed the content of the two URLs. Based on the structure and content of the articles, here's an enhanced style guide to better suit your needs for blog posts on data processing and scraping:

**Updated Style Guide for Effective Data-Focused Blog Writing**

- 1. Clarity and Precision:** Use clear, concise language that is easy to understand. Explain complex technical processes in a straightforward manner to ensure all readers, regardless of technical expertise, can follow.
- 2. Structured Format with Subheadings:** Organize content with informative subheadings that categorize information logically. This will guide readers smoothly through the content, making complex information accessible.
- 3. Detailed Step-by-Step Instructions:** When describing procedures like data cleaning or scraping, provide detailed, step-by-step instructions. This helps readers understand and replicate processes on their own.
- 4. Use Cases and Examples:** Incorporate practical examples and use cases to illustrate points. This helps bridge the gap between theory and real-world application.
- 5. Visual Elements:** Include diagrams, code snippets, and screenshots where relevant to visually support the text. Visual aids can enhance understanding of abstract concepts.
- 6. Highlight Common Pitfalls:** Address common mistakes and pitfalls to avoid in processes described, adding an educational layer to the content that can save readers time and frustration.
- 7. Ethical and Legal Considerations:** When applicable, discuss the ethical and legal considerations to keep in mind, particularly with data handling and privacy.
- 8. Engaging and Interactive Elements:** Encourage reader engagement by posing questions, inviting comments, or suggesting exercises they can try on their own.
- 9. Concise Summaries:** Conclude with a summary that recaps the key points discussed, reinforcing the learning and ensuring clarity.
- 10. Up-to-Date and Relevant:** Ensure content reflects the latest standards and practices in the field to keep it relevant and useful for readers.

Figure 3-9. ChatGPT refining a meta prompt by two URL web pages

Based on the style guide and the two blog posts, here's an improved copywriting style prompt:

To create a blog post for the client, adhere to the following guidelines:

1. Clarity and Directness: Use clear, direct language to explain complex

topics. Present information in an easy-to-understand manner. Aim for your audience to grasp concepts without needing prior knowledge.

...

**8. Use of Technical Terminology:** Use the appropriate technical language when discussing complex topics. However, make sure to provide definitions or explanations for these terms to keep the text accessible.

Model your writing after these blog post examples:

"What is an API? And How Do They Relate to Data Engineering?": This blog post uses clear and direct language to explain the concept of an API. It provides detailed explanations and uses practical examples of commonly used APIs. It links the technical concept of APIs to business growth and productivity. The information is well-categorized using subheadings like 'Types of API's.'

"How To Easily Setup A Google Cloud Project With APIs": This blog post is a practical guide on setting up a Google Cloud Project. It's broken down into numbered steps, each with a clear subheading. The post uses detailed explanations and includes practical examples such as screenshots. It also uses clear, direct language to guide the reader through the process.

Meta prompting offers a dynamic and innovative way to harness the power of generative AI models, fostering the creation of complex, multifaceted prompts and even prompts that generate other prompts. It broadens the application scope, from text and image generation to style and feature prompts, and optimization toward specific goals. As you continue to refine and explore the potential of meta prompting, it promises to be a game changer in how you utilize, interact with, and benefit from using LLMs.

## Summary

After reading this chapter, you are now aware of how crucial it is to give clear directions and examples to generate desired outputs. Also, you have hands-on experience extracting structured data from a hierarchical list using regular expressions in Python, and you've learned to utilize nested data structures like JSON and YAML to produce robust, parsable outputs.

You've learned several best practices and effective prompt engineering techniques, including the famous "Explain it like I'm five", role prompting, and meta prompting techniques. In the next chapter, you will learn how to use a popular LLM package called LangChain that'll help you to create more advanced prompt engineering workflows.

# Advanced Techniques for Text Generation with LangChain

Using simple prompt engineering techniques will often work for most tasks, but occasionally you'll need to use a more powerful toolkit to solve complex generative AI problems. Such problems and tasks include:

### *Context length*

Summarizing an entire book into a digestible synopsis.

### *Combining sequential LLM inputs/outputs*

Creating a story for a book including the characters, plot, and world building.

### *Performing complex reasoning tasks*

LLMs acting as an agent. For example, you could create an LLM agent to help you achieve your personal fitness goals.

To skillfully tackle such complex generative AI challenges, becoming acquainted with LangChain, an open source framework, is highly beneficial. This tool simplifies and enhances your LLM's workflows substantially.

## Introduction to LangChain

LangChain is a versatile framework that enables the creation of applications utilizing LLMs and is available as both a [Python](#) and a [TypeScript](#) package. Its central tenet is that the most impactful and distinct applications won't merely interface with a language model via an API, but will also:

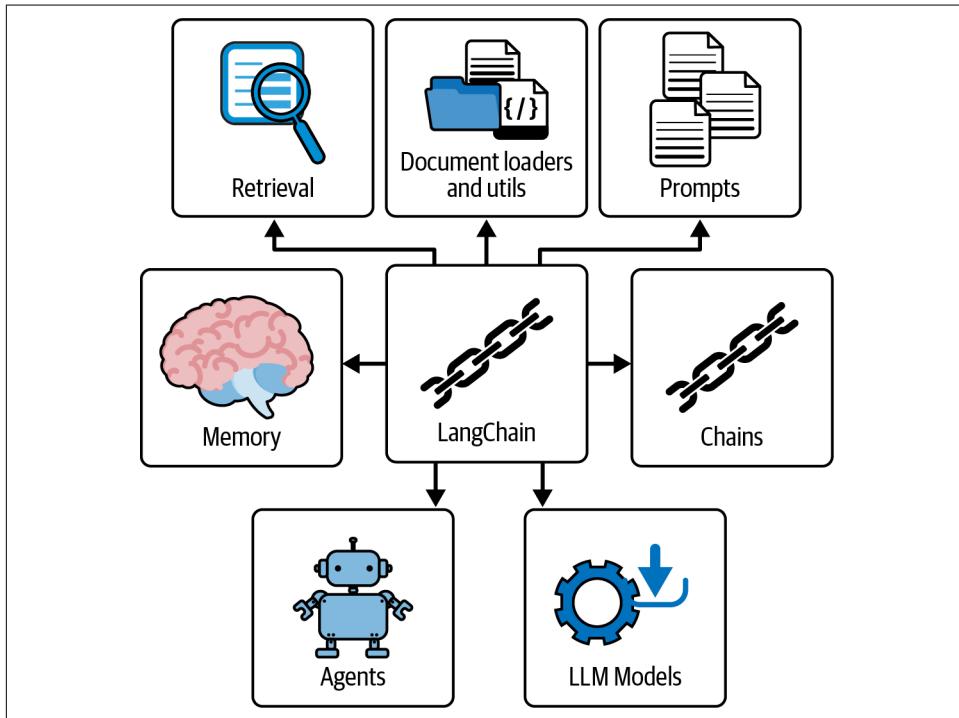
### *Enhance data awareness*

The framework aims to establish a seamless connection between a language model and external data sources.

### *Enhance agency*

It strives to equip language models with the ability to engage with and influence their environment.

The LangChain framework illustrated in [Figure 4-1](#) provides a range of modular abstractions that are essential for working with LLMs, along with a broad selection of implementations for these abstractions.



*Figure 4-1. The major modules of the LangChain LLM framework*

Each module is designed to be user-friendly and can be efficiently utilized independently or together. There are currently six common modules within LangChain:

#### *Model I/O*

Handles input/output operations related to the model

#### *Retrieval*

Focuses on retrieving relevant text for the LLM

### *Chains*

Also known as *LangChain runnables*, chains enable the construction of sequences of LLM operations or function calls

### *Agents*

Allows chains to make decisions on which tools to use based on high-level directives or instructions

### *Memory*

Persists the state of an application between different runs of a chain

### *Callbacks*

For running additional code on specific events, such as when every new token is generated

## Environment Setup

You can install LangChain on your terminal with either of these commands:

- `pip install langchain langchain-openai`
- `conda install -c conda-forge langchain langchain-openai`

If you would prefer to install the package requirements for the entire book, you can use the `requirements.txt` file from the GitHub repository.

It's recommended to install the packages within a virtual environment:

#### *Create a virtual environment*

```
python -m venv venv
```

#### *Activate the virtual environment*

```
source venv/bin/activate
```

#### *Install the dependencies*

```
pip install -r requirements.txt
```

LangChain requires integrations with one or more model providers. For example, to use OpenAI's model APIs, you'll need to install their Python package with `pip install openai`.

As discussed in [Chapter 1](#), it's best practice to set an environment variable called `OPENAI_API_KEY` in your terminal or load it from an `.env` file using `python-dotenv`. However, for prototyping you can choose to skip this step by passing in your API key directly when loading a chat model in LangChain:

```
from langchain_openai.chat_models import ChatOpenAI
chat = ChatOpenAI(api_key="api_key")
```



Hardcoding API keys in scripts is not recommended due to security reasons. Instead, utilize environment variables or configuration files to manage your keys.

In the constantly evolving landscape of LLMs, you can encounter the challenge of disparities across different model APIs. The lack of standardization in interfaces can induce extra layers of complexity in prompt engineering and obstruct the seamless integration of diverse models into your projects.

This is where LangChain comes into play. As a comprehensive framework, LangChain allows you to easily consume the varying interfaces of different models.

LangChain's functionality ensures that you aren't required to reinvent your prompts or code every time you switch between models. Its platform-agnostic approach promotes rapid experimentation with a broad range of models, such as [Anthropic](#), [Vertex AI](#), [OpenAI](#), and [BedrockChat](#). This not only expedites the model evaluation process but also saves critical time and resources by simplifying complex model integrations.

In the sections that follow, you'll be using the OpenAI package and their API in LangChain.

## Chat Models

Chat models such as GPT-4 have become the primary way to interface with OpenAI's API. Instead of offering a straightforward "input text, output text" response, they propose an interaction method where *chat messages* are the input and output elements.

Generating LLM responses using chat models involves inputting one or more messages into the chat model. In the context of LangChain, the currently accepted message types are `AIMessage`, `HumanMessage`, and `SystemMessage`. The output from a chat model will always be an `AIMessage`.

### *SystemMessage*

Represents information that should be instructions to the AI system. These are used to guide the AI's behavior or actions in some way.

### *HumanMessage*

Represents information coming from a human interacting with the AI system. This could be a question, a command, or any other input from a human user that the AI needs to process and respond to.

## *AI*Message

Represents information coming from the AI system itself. This is typically the AI's response to a `HumanMessage` or the result of a `SystemMessage` instruction.



Make sure to leverage the `SystemMessage` for delivering explicit directions. OpenAI has refined GPT-4 and upcoming LLM models to pay particular attention to the guidelines given within this type of message.

Let's create a joke generator in LangChain.

Input:

```
from langchain_openai.chat_models import ChatOpenAI
from langchain.schema import AIMessage, HumanMessage, SystemMessage

chat = ChatOpenAI(temperature=0.5)
messages = [SystemMessage(content='''Act as a senior software engineer
at a startup company.''''),
HumanMessage(content='''Please can you provide a funny joke
about software engineers?'''')]
response = chat.invoke(input=messages)
print(response.content)
```

Output:

```
Sure, here's a lighthearted joke for you:
Why did the software engineer go broke?
Because he lost his domain in a bet and couldn't afford to renew it.
```

First, you'll import `ChatOpenAI`, `AIMessage`, `HumanMessage`, and `SystemMessage`. Then create an instance of the `ChatOpenAI` class with a temperature parameter of 0.5 (randomness).

After creating a model, a list named `messages` is populated with a `SystemMessage` object, defining the role for the LLM, and a `HumanMessage` object, which asks for a software engineer—related joke.

Calling the chat model with `.invoke(input=messages)` feeds the LLM with a list of messages, and then you retrieve the LLM's response with `response.content`.

There is a legacy method that allows you to directly call the `chat` object with `chat(messages=messages)`:

```
response = chat(messages=messages)
```

## Streaming Chat Models

You might have observed while using ChatGPT how words are sequentially returned to you, one character at a time. This distinct pattern of response generation is referred to as *streaming*, and it plays a crucial role in enhancing the performance of chat-based applications:

```
for chunk in chat.stream(messages):
    print(chunk.content, end="", flush=True)
```

When you call `chat.stream(messages)`, it yields chunks of the message one at a time. This means each segment of the chat message is individually returned. As each chunk arrives, it is then instantaneously printed to the terminal and flushed. This way, *streaming* allows for minimal latency from your LLM responses.

Streaming holds several benefits from an end-user perspective. First, it dramatically reduces the waiting time for users. As soon as the text starts generating character by character, users can start interpreting the message. There's no need for a full message to be constructed before it is seen. This, in turn, significantly enhances user interactivity and minimizes latency.

Nevertheless, this technique comes with its own set of challenges. One significant challenge is parsing the outputs while they are being streamed. Understanding and appropriately responding to the message as it is being formed can prove to be intricate, especially when the content is complex and detailed.

## Creating Multiple LLM Generations

There may be scenarios where you find it useful to generate multiple responses from LLMs. This is particularly true while creating dynamic content like social media posts. Rather than providing a list of messages, you provide a *list of message lists*.

Input:

```
# 2x lists of messages, which is the same as [messages, messages]
synchronous_llm_result = chat.batch([messages]*2)
print(synchronous_llm_result)
```

Output:

```
[AIMessage(content='''Sure, here's a lighthearted joke for you:\n\nWhy did
the software engineer go broke?\n\nBecause he kept forgetting to Ctrl+ Z
his expenses!''' ),
AIMessage(content='''Sure, here's a lighthearted joke for you:\n\nWhy do
software engineers prefer dark mode?\n\nBecause it's easier on their
"byte" vision!''' )]
```

The benefit of using `.batch()` over `.invoke()` is that you can parallelize the number of API requests made to OpenAI.

For any runnable in LangChain, you can add a `RunnableConfig` argument to the `batch` function that contains many configurable parameters, including `max_concurrency`:

```
from langchain_core.runnables.config import RunnableConfig

# Create a RunnableConfig with the desired concurrency limit:
config = RunnableConfig(max_concurrency=5)

# Call the .batch() method with the inputs and config:
results = chat.batch([messages, messages], config=config)
```



In computer science, *asynchronous (async) functions* are those that operate independently of other processes, thereby enabling several API requests to be run concurrently without waiting for each other. In LangChain, these async functions let you make many API requests all at once, not one after the other. This is especially helpful in more complex workflows and decreases the overall latency to your users.

Most of the asynchronous functions within LangChain are simply prefixed with the letter `a`, such as `.ainvoke()` and `.abatch()`. If you would like to use the `async` API for more efficient task performance, then utilize these functions.

## LangChain Prompt Templates

Up until this point, you've been hardcoding the strings in the `ChatOpenAI` objects. As your LLM applications grow in size, it becomes increasingly important to utilize *prompt templates*.

*Prompt templates* are good for generating reproducible prompts for AI language models. They consist of a *template*, a text string that can take in parameters, and construct a text prompt for a language model.

Without prompt templates, you would likely use Python `f-string` formatting:

```
language = "Python"
prompt = f"What is the best way to learn coding in {language}?"
print(prompt) # What is the best way to learn coding in Python?
```

But why not simply use an `f-string` for prompt templating? Using LangChain's prompt templates instead allows you to easily:

- Validate your prompt inputs
- Combine multiple prompts together with composition
- Define custom selectors that will inject k-shot examples into your prompt

- Save and load prompts from `.yml` and `.json` files
- Create custom prompt templates that execute additional code or instructions when created

## LangChain Expression Language (LCEL)

The `|` pipe operator is a key component of LangChain Expression Language (LCEL) that allows you to chain together different components or *runnables* in a data processing pipeline.

In LCEL, the `|` operator is similar to the Unix pipe operator. It takes the output of one component and feeds it as input to the next component in the chain. This allows you to easily connect and combine different components to create a complex chain of operations:

```
chain = prompt | model
```

The `|` operator is used to chain together the prompt and model components. The output of the prompt component is passed as input to the model component. This chaining mechanism allows you to build complex chains from basic components and enables the seamless flow of data between different stages of the processing pipeline.

Additionally, *the order matters*, so you could technically create this chain:

```
bad_order_chain = model | prompt
```

But it would produce an error after using the `invoke` function, because the values returned from `model` are not compatible with the expected inputs for the prompt.

Let's create a business name generator using prompt templates that will return five to seven relevant business names:

```
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.prompts import (SystemMessagePromptTemplate,
ChatPromptTemplate)

template = """
You are a creative consultant brainstorming names for businesses.

You must follow the following principles:
{principles}

Please generate a numerical list of five catchy names for a start-up in the
{industry} industry that deals with {context}?

Here is an example of the format:
1. Name1
2. Name2
3. Name3
4. Name4
```

```

5. Name5
"""

model = ChatOpenAI()
system_prompt = SystemMessagePromptTemplate.from_template(template)
chat_prompt = ChatPromptTemplate.from_messages([system_prompt])

chain = chat_prompt | model

result = chain.invoke({
    "industry": "medical",
    "context": '''creating AI solutions by automatically summarizing patient
records''',
    "principles":'''1. Each name should be short and easy to
remember. 2. Each name should be easy to pronounce.
3. Each name should be unique and not already taken by another company.'''
})

print(result.content)

```

Output:

1. SummarAI
2. MediSummar
3. AutoDocs
4. RecordAI
5. SmartSummarize

First, you'll import `ChatOpenAI`, `SystemMessagePromptTemplate`, and `ChatPromptTemplate`. Then, you'll define a prompt template with specific guidelines under `template`, instructing the LLM to generate business names. `ChatOpenAI()` initializes the chat, while `SystemMessagePromptTemplate.from_template(template)` and `ChatPromptTemplate.from_messages([system_prompt])` create your prompt template.

You create an LCEL chain by piping together `chat_prompt` and the `model`, which is then *invoked*. This replaces the `{industries}`, `{context}`, and `{principles}` place-holders in the prompt with the dictionary values within the `invoke` function.

Finally, you extract the LLM's response as a string accessing the `.content` property on the `result` variable.



### Give Direction and Specify Format

Carefully crafted instructions might include things like “You are a creative consultant brainstorming names for businesses” and “Please generate a numerical list of five to seven catchy names for a start-up.” Cues like these guide your LLM to perform the exact task you require from it.

# Using PromptTemplate with Chat Models

LangChain provides a more traditional template called `PromptTemplate`, which requires `input_variables` and `template` arguments.

Input:

```
from langchain_core.prompts import PromptTemplate
from langchain.prompts.chat import SystemMessagePromptTemplate
from langchain_openai.chat_models import ChatOpenAI
prompt = PromptTemplate(
    template='''You are a helpful assistant that translates {input_language} to
{output_language}.'''',
    input_variables=['input_language', 'output_language'],
)
system_message_prompt = SystemMessagePromptTemplate(prompt=prompt)
chat = ChatOpenAI()
chat.invoke(system_message_prompt.format_messages(
    input_language="English", output_language="French"))
```

Output:

```
AIMessage(content="Vous êtes un assistant utile qui traduit l'anglais en
français.", additional_kwargs={}, example=False)
```

# Output Parsers

In [Chapter 3](#), you used regular expressions (regex) to extract structured data from text that contained numerical lists, but it's possible to do this automatically in LangChain with *output parsers*.

*Output parsers* are a higher-level abstraction provided by LangChain for parsing structured data from LLM string responses. Currently the available output parsers are:

## *List parser*

Returns a list of comma-separated items.

## *Datetime parser*

Parses an LLM output into datetime format.

## *Enum parser*

Parses strings into enum values.

## *Auto-fixing parser*

Wraps another output parser, and if that output parser fails, it will call another LLM to fix any errors.

## *Pydantic (JSON) parser*

Parses LLM responses into JSON output that conforms to a Pydantic schema.

### *Retry parser*

Provides retrying a failed parse from a previous output parser.

### *Structured output parser*

Can be used when you want to return multiple fields.

### *XML parser*

Parses LLM responses into an XML-based format.

As you'll discover, there are two important functions for LangChain output parsers:

`.get_format_instructions()`

This function provides the necessary instructions into your prompt to output a structured format that can be parsed.

`.parse(llm_output: str)`

This function is responsible for parsing your LLM responses into a predefined format.

Generally, you'll find that the Pydantic (JSON) parser with `ChatOpenAI()` provides the most flexibility.

The Pydantic (JSON) parser takes advantage of the [Pydantic](#) library in Python. Pydantic is a data validation library that provides a way to validate incoming data using Python type annotations. This means that Pydantic allows you to create schemas for your data and automatically validates and parses input data according to those schemas.

Input:

```
from langchain_core.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
)
from langchain_openai.chat_models import ChatOpenAI
from langchain.output_parsers import PydanticOutputParser
from pydantic.v1 import BaseModel, Field
from typing import List

temperature = 0.0

class BusinessName(BaseModel):
    name: str = Field(description="The name of the business")
    rating_score: float = Field(description='''The rating score of the
    business. 0 is the worst, 10 is the best.''')

class BusinessNames(BaseModel):
    names: List[BusinessName] = Field(description='''A list
    of business names''')


# Set up a parser + inject instructions into the prompt template:
```

```

parser = PydanticOutputParser(pydantic_object=BusinessNames)

principles = """
- The name must be easy to remember.
- Use the {industry} industry and Company context to create an effective name.
- The name must be easy to pronounce.
- You must only return the name without any other text or characters.
- Avoid returning full stops, \n, or any other characters.
- The maximum length of the name must be 10 characters.
"""

# Chat Model Output Parser:
model = ChatOpenAI()
template = """Generate five business names for a new start-up company in the
{industry} industry.
You must follow the following principles: {principles}
{format_instructions}
"""
system_message_prompt = SystemMessagePromptTemplate.from_template(template)
chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt])

# Creating the LCEL chain:
prompt_and_model = chat_prompt | model

result = prompt_and_model.invoke(
    {
        "principles": principles,
        "industry": "Data Science",
        "format_instructions": parser.get_format_instructions(),
    }
)
# The output parser, parses the LLM response into a Pydantic object:
print(parser.parse(result.content))

```

Output:

```

names=[BusinessName(name='DataWiz', rating_score=8.5),
BusinessName(name='InsightIQ',
rating_score=9.2), BusinessName(name='AnalytiQ', rating_score=7.8),
BusinessName(name='SciData', rating_score=8.1),
BusinessName(name='InfoMax', rating_score=9.5)]

```

After you've loaded the necessary libraries, you'll set up a ChatOpenAI model. Then create SystemMessagePromptTemplate from your template and form a ChatPrompt Template with it. You'll use the Pydantic models BusinessName and BusinessNames to structure your desired output, a list of unique business names. You'll create a Pydantic parser for parsing these models and format the prompt using user-inputted variables by calling the invoke function. Feeding this customized prompt to your model, you're enabling it to produce creative, unique business names by using the parser.

It's possible to use output parsers inside of LCEL by using this syntax:

```
chain = prompt | model | output_parser
```

Let's add the output parser directly to the chain.

Input:

```
parser = PydanticOutputParser(pydantic_object=BusinessNames)
chain = chat_prompt | model | parser

result = chain.invoke(
{
    "principles": principles,
    "industry": "Data Science",
    "format_instructions": parser.get_format_instructions(),
}
)
print(result)
```

Output:

```
names=[BusinessName(name='DataTech', rating_score=9.5),...]
```

The chain is now responsible for prompt formatting, LLM calling, and parsing the LLM's response into a Pydantic object.



### Specify Format

The preceding prompts use Pydantic models and output parsers, allowing you explicitly tell an LLM your desired response format.

It's worth knowing that by asking an LLM to provide structured JSON output, you can create a flexible and generalizable API from the LLM's response. There are limitations to this, such as the size of the JSON created and the reliability of your prompts, but it still is a promising area for LLM applications.



You should take care of edge cases as well as adding error handling statements, since LLM outputs might not always be in your desired format.

Output parsers save you from the complexity and intricacy of regular expressions, providing easy-to-use functionalities for a variety of use cases. Now that you've seen them in action, you can utilize output parsers to effortlessly structure and retrieve the data you need from an LLM's output, harnessing the full potential of AI for your tasks.

Furthermore, using parsers to structure the data extracted from LLMs allows you to easily choose how to organize outputs for more efficient use. This can be useful if you're dealing with extensive lists and need to sort them by certain criteria, like business names.

## LangChain Evals

As well as output parsers to check for formatting errors, most AI systems also make use of *evals*, or evaluation metrics, to measure the performance of each prompt response. LangChain has a number of off-the-shelf evaluators, which can be directly be logged in their [LangSmith](#) platform for further debugging, monitoring, and testing. [Weights and Biases](#) is alternative machine learning platform that offers similar functionality and tracing capabilities for LLMs.

Evaluation metrics are useful for more than just prompt testing, as they can be used to identify positive and negative examples for retrieval as well as to build datasets for fine-tuning custom models.

Most eval metrics rely on a set of test cases, which are input and output pairings where you know the correct answer. Often these reference answers are created or curated manually by a human, but it's also common practice to use a smarter model like GPT-4 to generate the ground truth answers, which has been done for the following example. Given a list of descriptions of financial transactions, we used GPT-4 to classify each transaction with a `transaction_category` and `transaction_type`. The process can be found in the `langchain-evals.ipynb` Jupyter Notebook in the [GitHub repository](#) for the book.

With the GPT-4 answer being taken as the correct answer, it's now possible to rate the accuracy of smaller models like GPT-3.5-turbo and Mixtral 8x7b (called `mistral-small` in the API). If you can achieve good enough accuracy with a smaller model, you can save money or decrease latency. In addition, if that model is available open source like [Mistral's model](#), you can migrate that task to run on your own servers, avoiding sending potentially sensitive data outside of your organization. We recommend testing with an external API first, before going to the trouble of self-hosting an OS model.

Remember to [sign up](#) and subscribe to obtain an API key; then expose that as an environment variable by typing in your terminal:

```
export MISTRAL_API_KEY=api-key
```

The following script is part of a [notebook](#) that has previously defined a dataframe `df`. For brevity let's investigate only the evaluation section of the script, assuming a dataframe is already defined.

Input:

```
import os
from langchain_mistralai.chat_models import ChatMistralAI
from langchain.output_parsers import PydanticOutputParser
from langchain_core.prompts import ChatPromptTemplate
from pydantic.v1 import BaseModel
from typing import Literal, Union
from langchain_core.output_parsers import StrOutputParser

# 1. Define the model:
mistral_api_key = os.environ["MISTRAL_API_KEY"]

model = ChatMistralAI(model="mistral-small", mistral_api_key=mistral_api_key)

# 2. Define the prompt:
system_prompt = """You are an expert at analyzing
bank transactions, you will be categorizing a single
transaction.
Always return a transaction type and category:
do not return None.
Format Instructions:
{format_instructions}"""

user_prompt = """Transaction Text:
{transaction}"""

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            system_prompt,
        ),
        (
            "user",
            user_prompt,
        ),
    ],
)

# 3. Define the pydantic model:
class EnrichedTransactionInformation(BaseModel):
    transaction_type: Union[
        Literal["Purchase", "Withdrawal", "Deposit",
               "Bill Payment", "Refund"], None
    ]
    transaction_category: Union[
        Literal["Food", "Entertainment", "Transport",
               "Utilities", "Rent", "Other"], None
    ]
```

```

# 4. Define the output parser:
output_parser = PydanticOutputParser(
    pydantic_object=EnrichedTransactionInformation)

# 5. Define a function to try to fix and remove the backslashes:
def remove_back_slashes(string):
    # double slash to escape the slash
    cleaned_string = string.replace("\\\\", "")
    return cleaned_string

# 6. Create an LCEL chain that fixes the formatting:
chain = prompt | model | StrOutputParser() \
| remove_back_slashes | output_parser

transaction = df.iloc[0]["Transaction Description"]
result = chain.invoke(
    {
        "transaction": transaction,
        "format_instructions": \
            output_parser.get_format_instructions(),
    }
)

# 7. Invoke the chain for the whole dataset:
results = []

for i, row in tqdm(df.iterrows(), total=len(df)):
    transaction = row["Transaction Description"]
    try:
        result = chain.invoke(
            {
                "transaction": transaction,
                "format_instructions": \
                    output_parser.get_format_instructions(),
            }
        )
    except:
        result = EnrichedTransactionInformation(
            transaction_type=None,
            transaction_category=None
        )
    results.append(result)

# 8. Add the results to the dataframe, as columns transaction type and
# transaction category:
transaction_types = []
transaction_categories = []

for result in results:
    transaction_types.append(result.transaction_type)
    transaction_categories.append(

```

```
        result.transaction_category)

df["mistral_transaction_type"] = transaction_types
df["mistral_transaction_category"] = transaction_categories
df.head()
```

Output:

```
Transaction Description transaction_type
transaction_category    mistral_transaction_type
mistral_transaction_category
0      cash deposit at local branch    Deposit Other    Deposit
Other
1      cash deposit at local branch    Deposit Other    Deposit
Other
2      withdrew money for rent payment Withdrawal      Rent
Withdrawal      Rent
3      withdrew cash for weekend expenses      Withdrawal      Other
Withdrawal      Other
4      purchased books from the bookstore      Purchase      Other
Purchase      Entertainment
```

The code does the following:

1. `from langchain_mistralai.chat_models import ChatMistralAI`: We import LangChain's Mistral implementation.
2. `from langchain.output_parsers import PydanticOutputParser`: Imports the PydanticOutputParser class, which is used for parsing output using Pydantic models. We also import a string output parser to handle an interim step where we remove backslashes from the JSON key (a common problem with responses from Mistral).
3. `mistral_api_key = os.environ["MISTRAL_API_KEY"]`: Retrieves the Mistral API key from the environment variables. This needs to be set prior to running the notebook.
4. `model = ChatMistralAI(model="mistral-small", mistral_api_key=mistral_api_key)`: Initializes an instance of ChatMistralAI with the specified model and API key. Mistral Small is what they call the Mixtral 8x7b model (also available open source) in their API.
5. `system_prompt` and `user_prompt`: These lines define templates for the system and user prompts used in the chat to classify the transactions.
6. `class EnrichedTransactionInformation(BaseModel)`: Defines a Pydantic model EnrichedTransactionInformation with two fields: `transaction_type` and `transaction_category`, each with specific allowed values and the possibility of being `None`. This is what tells us if the output is in the correct format.

7. `def remove_back_slashes(string):` Defines a function to remove backslashes from a string.
8. `chain = prompt | model | StrOutputParser() | remove_back_slashes | output_parser:` Updates the chain to include a string output parser and the `remove_back_slashes` function before the original output parser.
9. `transaction = df.iloc[0]["Transaction Description"]:` Extracts the first transaction description from a dataframe `df`. This dataframe is loaded earlier in the [Jupyter Notebook](#) (omitted for brevity).
10. `for i, row in tqdm(df.iterrows(), total=len(df)):` Iterates over each row in the dataframe `df`, with a progress bar.
11. `result = chain.invoke(...):` Inside the loop, the chain is invoked for each transaction.
12. `except:` In case of an exception, a default `EnrichedTransactionInformation` object with `None` values is created. These will be treated as errors in evaluation but will not break the processing loop.
13. `df["mistral_transaction_type"] = transaction_types, df["mistral_transaction_category"] = transaction_categories:` Adds the transaction types and categories as new columns in the dataframe, which we then display with `df.head()`.

With the responses from Mistral saved in the dataframe, it's possible to compare them to the transaction categories and types defined earlier to check the accuracy of Mistral. The most basic LangChain eval metric is to do an exact string match of a prediction against a reference answer, which returns a score of 1 if correct, and a 0 if incorrect. The notebook gives an example of how to [implement this](#), which shows that Mistral's accuracy is 77.5%. However, if all you are doing is comparing strings, you probably don't need to implement it in LangChain.

Where LangChain is valuable is in its standardized and tested approaches to implementing more advanced evaluators using LLMs. The evaluator `labeled_pairwise_string` compares two outputs and gives a reason for choosing between them, using GPT-4. One common use case for this type of evaluator is to compare the outputs from two different prompts or models, particularly if the models being tested are less sophisticated than GPT-4. This evaluator using GPT-4 does still work for evaluating GPT-4 responses, but you should manually review the reasoning and scores to ensure it is doing a good job: if GPT-4 is bad at a task, it may also be bad at evaluating that task. In [the notebook](#), the same transaction classification was run again with the model changed to `model = ChatOpenAI(model="gpt-3.5-turbo-1106", model_kwargs={"response_format": {"type": "json_object"}},)`. Now it's possible to do pairwise comparison between the Mistral and GPT-3.5 responses, as shown

in the following example. You can see in the output the reasoning that is given to justify the score.

Input:

```
# Evaluate answers using LangChain evaluators:
from langchain.evaluation import load_evaluator
evaluator = load_evaluator("labeled_pairwise_string")

row = df.iloc[0]
transaction = row["Transaction Description"]
gpt3pt5_category = row["gpt3.5_transaction_category"]
gpt3pt5_type = row["gpt3.5_transaction_type"]
mistral_category = row["mistral_transaction_category"]
mistral_type = row["mistral_transaction_type"]
reference_category = row["transaction_category"]
reference_type = row["transaction_type"]

# Put the data into JSON format for the evaluator:
gpt3pt5_data = f"""{
    "transaction_category": "{gpt3pt5_category}",
    "transaction_type": "{gpt3pt5_type}"
}"""

mistral_data = f"""{
    "transaction_category": "{mistral_category}",
    "transaction_type": "{mistral_type}"
}"""

reference_data = f"""{
    "transaction_category": "{reference_category}",
    "transaction_type": "{reference_type}"
}"""

# Set up the prompt input for context for the evaluator:
input_prompt = """You are an expert at analyzing bank
transactions,
you will be categorizing a single transaction.
Always return a transaction type and category: do not
return None.
Format Instructions:
{format_instructions}
Transaction Text:
{transaction}
"""

transaction_types.append(transaction_type_score)
transaction_categories.append(
    transaction_category_score)

accuracy_score = 0
```

```

for transaction_type_score, transaction_category_score \
    in zip(
        transaction_types, transaction_categories
    ):
    accuracy_score += transaction_type_score['score'] + \
        transaction_category_score['score']

accuracy_score = accuracy_score / (len(transaction_types) \
    * 2)
print(f"Accuracy score: {accuracy_score}")

evaluator.evaluate_string_pairs(
    prediction=gpt3pt5_data,
    prediction_b=mistral_data,
    input=input_prompt.format(
        format_instructions=output_parser.get_format_instructions(),
        transaction=transaction),
    reference=reference_data,
)

```

Output:

```

{'reasoning': '''Both Assistant A and Assistant B provided the exact same
response to the user\'s question. Their responses are both helpful, relevant,
correct, and demonstrate depth of thought. They both correctly identified the
transaction type as "Deposit" and the transaction category as "Other" based on
the transaction text provided by the user. Both responses are also
well-formatted according to the JSON schema provided by the user. Therefore,
it\'s a tie between the two assistants. \n\nFinal Verdict: [[C]]''',
 'value': None,
 'score': 0.5}

```

This code demonstrates the simple exact string matching evaluator from LangChain:

1. `evaluator = load_evaluator("labeled_pairwise_string")`: This is a helper function that can be used to load any LangChain evaluator by name. In this case, it is the `labeled_pairwise_string` evaluator being used.
2. `row = df.iloc[0]`: This line and the seven lines that follow get the first row and extract the values for the different columns needed. It includes the transaction description, as well as the Mistral and GPT-3.5 transaction category and types. This is showcasing a single transaction, but this code can easily run in a loop through each transaction, replacing this line with an `iterrows` function for `i, row in tqdm(df.iterrows(), total=len(df))`, as is done later in [the notebook](#).
3. `gpt3pt5_data = f"""\{{}`: To use the pairwise comparison evaluator, we need to pass the results in a way that is formatted correctly for the prompt. This is done for Mistral and GPT-3.5, as well as the reference data.

4. `input_prompt = """You are an expert...:` The other formatting we have to get right is in the prompt. To get accurate evaluation scores, the evaluator needs to see the instructions that were given for the task.
5. `evaluator.evaluate_string_pairs(...:` All that remains is to run the evaluator by passing in the `prediction` and `prediction_b` (GPT-3.5 and Mistral, respectively), as well as the `input` prompt, and `reference` data, which serves as the ground truth.
6. Following this code [in the notebook](#), there is an example of looping through and running the evaluator on every row in the dataframe and then saving the results and reasoning back to the dataframe.

This example demonstrates how to use a LangChain evaluator, but there are many different kinds of evaluator available. String distance ([Levenshtein](#)) or [embedding distance](#) evaluators are often used in scenarios where answers are not an exact match for the reference answer, but only need to be close enough semantically. Levenshtein distance allows for fuzzy matches based on how many single-character edits would be needed to transform the predicted text into the reference text, and embedding distance makes use of vectors (covered in [Chapter 5](#)) to calculate similarity between the answer and reference.

The other kind of evaluator we often use in our work is pairwise comparisons, which are useful for comparing two different prompts or models, using a smarter model like GPT-4. This type of comparison is helpful because reasoning is provided for each comparison, which can be useful in debugging why one approach was favored over another. The [notebook for this section](#) shows an example of using a pairwise comparison evaluator to check GPT-3.5-turbo's accuracy versus Mixtral 8x7b.



### Evaluate Quality

Without defining an appropriate set of eval metrics to define success, it can be difficult to tell if changes to the prompt or wider system are improving or harming the quality of responses. If you can automate eval metrics using smart models like GPT-4, you can iterate faster to improve results without costly or time-consuming manual human review.

# OpenAI Function Calling

*Function calling* provides an alternative method to output parsers, leveraging fine-tuned OpenAI models. These models identify when a function should be executed and generate a JSON response with the *name* and *arguments* for a predefined function. Several use cases include:

## *Designing sophisticated chat bots*

Capable of organizing and managing schedules. For example, you can define a function to schedule a meeting: `schedule_meeting(date: str, time: str, attendees: List[str])`.

## *Convert natural language into actionable API calls*

A command like “Turn on the hallway lights” can be converted to `control_device(device: str, action: 'on' | 'off')` for interacting with your home automation API.

## *Extracting structured data*

This could be done by defining a function such as `extract_contextual_data(context: str, data_points: List[str])` or `search_database(query: str)`.

Each function that you use within function calling will require an appropriate *JSON schema*. Let’s explore an example with the OpenAI package:

```
from openai import OpenAI
import json
from os import getenv

def schedule_meeting(date, time, attendees):
    # Connect to calendar service:
    return { "event_id": "1234", "status": "Meeting scheduled successfully!",
             "date": date, "time": time, "attendees": attendees }

OPENAI_FUNCTIONS = {
    "schedule_meeting": schedule_meeting
}
```

After importing OpenAI and json, you’ll create a function named `schedule_meeting`. This function is a mock-up, simulating the process of scheduling a meeting, and returns details such as `event_id`, `date`, `time`, and `attendees`. Following that, make an `OPENAI_FUNCTIONS` dictionary to map the function name to the actual function for ease of reference.

Next, define a `functions` list that provides the function's JSON schema. This schema includes its name, a brief description, and the parameters it requires, guiding the LLM on how to interact with it:

```
# Our predefined function JSON schema:  
functions = [  
    {  
        "type": "function",  
        "function": {  
            "type": "object",  
            "name": "schedule_meeting",  
            "description": '''Set a meeting at a specified date and time for  
designated attendees''',  
            "parameters": {  
                "type": "object",  
                "properties": {  
                    "date": {"type": "string", "format": "date"},  
                    "time": {"type": "string", "format": "time"},  
                    "attendees": {"type": "array", "items": {"type": "string"}},  
                },  
                "required": ["date", "time", "attendees"],  
            },  
        },  
    },  
]
```



### Specify Format

When using function calling with your OpenAI models, always ensure to define a detailed JSON schema (including the name and description). This acts as a blueprint for the function, guiding the model to understand when and how to properly invoke it.

After defining the functions, let's make an OpenAI API request. Set up a `messages` list with the user query. Then, using an OpenAI `client` object, you'll send this message and the function schema to the model. The LLM analyzes the conversation, discerns a need to trigger a function, and provides the function name and arguments. The `function` and `function_args` are parsed from the LLM response. Then the function is executed, and its results are added back into the conversation. Then you call the model again for a user-friendly summary of the entire process.

Input:

```
client = OpenAI(api_key=getenv("OPENAI_API_KEY"))  
  
# Start the conversation:  
messages = [  
    {  
        "role": "user",  
        "content": '''Schedule a meeting on 2023-11-01 at 14:00
```

```

        with Alice and Bob.''",
    }
]

# Send the conversation and function schema to the model:
response = client.chat.completions.create(
    model="gpt-3.5-turbo-1106",
    messages=messages,
    tools=functions,
)
response = response.choices[0].message

# Check if the model wants to call our function:
if response.tool_calls:
    # Get the first function call:
    first_tool_call = response.tool_calls[0]

    # Find the function name and function args to call:
    function_name = first_tool_call.function.name
    function_args = json.loads(first_tool_call.function.arguments)
    print("This is the function name: ", function_name)
    print("These are the function arguments: ", function_args)

    function = OPENAI_FUNCTIONS.get(function_name)

    if not function:
        raise Exception(f"Function {function_name} not found.")

    # Call the function:
    function_response = function(**function_args)

# Share the function's response with the model:
messages.append(
    {
        "role": "function",
        "name": "schedule_meeting",
        "content": json.dumps(function_response),
    }
)

# Let the model generate a user-friendly response:
second_response = client.chat.completions.create(
    model="gpt-3.5-turbo-0613", messages=messages
)
print(second_response.choices[0].message.content)

```

Output:

```
These are the function arguments: {'date': '2023-11-01', 'time': '14:00',  
'attendees': ['Alice', 'Bob']}  
This is the function name: schedule_meeting  
I have scheduled a meeting on 2023-11-01 at 14:00 with Alice and Bob.  
The event ID is 1234.
```

Several important points to note while function calling:

- It's possible to have many functions that the LLM can call.
- OpenAI can hallucinate function parameters, so be more explicit within the system message to overcome this.
- The `function_call` parameter can be set in various ways:
  - To mandate a specific function call: `tool_choice: {"type": "function", "function": {"name": "my_function"}}}`.
  - For a user message without function invocation: `tool_choice: "none"`.
  - By default (`tool_choice: "auto"`), the model autonomously decides if and which function to call.

## Parallel Function Calling

You can set your chat messages to include intents that request simultaneous calls to multiple tools. This strategy is known as *parallel function calling*.

Modifying the previously used code, the `messages` list is updated to mandate the scheduling of two meetings:

```
# Start the conversation:  
messages = [  
    {  
        "role": "user",  
        "content": '''Schedule a meeting on 2023-11-01 at 14:00 with Alice  
and Bob. Then I want to schedule another meeting on 2023-11-02 at  
15:00 with Charlie and Dave.'''  
    }  
]
```

Then, adjust the previous code section by incorporating a `for` loop.

Input:

```
# Send the conversation and function schema to the model:  
response = client.chat.completions.create(  
    model="gpt-3.5-turbo-1106",  
    messages=messages,  
    tools=functions,  
)
```

```

response = response.choices[0].message

# Check if the model wants to call our function:
if response.tool_calls:
    for tool_call in response.tool_calls:
        # Get the function name and arguments to call:
        function_name = tool_call.function.name
        function_args = json.loads(tool_call.function.arguments)
        print("This is the function name: ", function_name)
        print("These are the function arguments: ", function_args)

        function = OPENAI_FUNCTIONS.get(function_name)

        if not function:
            raise Exception(f"Function {function_name} not found.")

        # Call the function:
        function_response = function(**function_args)

        # Share the function's response with the model:
        messages.append(
            {
                "role": "function",
                "name": function_name,
                "content": json.dumps(function_response),
            }
        )

    # Let the model generate a user-friendly response:
    second_response = client.chat.completions.create(
        model="gpt-3.5-turbo-0613", messages=messages
    )

    print(second_response.choices[0].message.content)

```

Output:

```

This is the function name: schedule_meeting
These are the function arguments: {'date': '2023-11-01', 'time': '14:00',
'attendees': ['Alice', 'Bob']}
This is the function name: schedule_meeting
These are the function arguments: {'date': '2023-11-02', 'time': '15:00',
'attendees': ['Charlie', 'Dave']}
Two meetings have been scheduled:
1. Meeting with Alice and Bob on 2023-11-01 at 14:00.
2. Meeting with Charlie and Dave on 2023-11-02 at 15:00.

```

From this example, it's clear how you can effectively manage multiple function calls. You've seen how the `schedule_meeting` function was called twice in a row to arrange different meetings. This demonstrates how flexibly and effortlessly you can handle varied and complex requests using AI-powered tools.

# Function Calling in LangChain

If you'd prefer to avoid writing JSON schema and simply want to extract structured data from an LLM response, then LangChain allows you to use function calling with Pydantic.

Input:

```
from langchain.output_parsers.openai_tools import PydanticToolsParser
from langchain_core.utils.function_calling import convert_to_openai_tool
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.pydantic_v1 import BaseModel, Field
from typing import Optional

class Article(BaseModel):
    """Identifying key points and contrarian views in an article."""

    points: str = Field(..., description="Key points from the article")
    contrarian_points: Optional[str] = Field(
        None, description="Any contrarian points acknowledged in the article"
    )
    author: Optional[str] = Field(None, description="Author of the article")

    _EXTRACTION_TEMPLATE = """Extract and save the relevant entities mentioned \
in the following passage together with their properties.

If a property is not present and is not required in the function parameters,
do not include it in the output."""

    # Create a prompt telling the LLM to extract information:
    prompt = ChatPromptTemplate.from_messages(
        {("system", _EXTRACTION_TEMPLATE), ("user", "{input}")})
    )

    model = ChatOpenAI()

    pydantic_schemas = [Article]

    # Convert Pydantic objects to the appropriate schema:
    tools = [convert_to_openai_tool(p) for p in pydantic_schemas]

    # Give the model access to these tools:
    model = model.bind_tools(tools=tools)

    # Create an end to end chain:
    chain = prompt | model | PydanticToolsParser(tools=pydantic_schemas)

    result = chain.invoke(
        {
            "input": """In the recent article titled 'AI adoption in industry,'
key points addressed include the growing interest ... However, the
```

```
        author, Dr. Jane Smith, ..."""
    }
)
print(result)
```

Output:

```
[Article(points='The growing interest in AI in various sectors, ...',
contrarian_points='Without stringent regulations, ...',
author='Dr. Jane Smith')]
```

You'll start by importing various modules, including PydanticToolsParser and ChatPromptTemplate, essential for parsing and templating your prompts. Then, you'll define a Pydantic model, Article, to specify the structure of the information you want to extract from a given text. With the use of a custom prompt template and the ChatOpenAI model, you'll instruct the AI to extract key points and contrarian views from an article. Finally, the extracted data is neatly converted into your predefined Pydantic model and printed out, allowing you to see the structured information pulled from the text.

There are several key points, including:

*Converting Pydantic schema to OpenAI tools*

```
tools = [convert_to_openai_tool(p) for p in pydantic_schemas]
```

*Binding the tools directly to the LLM*

```
model = model.bind_tools(tools=tools)
```

*Creating an LCEL chain that contains a tools parser*

```
chain = prompt | model | PydanticToolsParser(tools=pydantic_schemas)
```

## Extracting Data with LangChain

The `create_extraction_chain_pydantic` function provides a more concise version of the previous implementation. By simply inserting a Pydantic model and an LLM that supports function calling, you can easily achieve parallel function calling.

Input:

```
from langchain.chains.openai_tools import create_extraction_chain_pydantic
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.pydantic_v1 import BaseModel, Field

# Make sure to use a recent model that supports tools:
model = ChatOpenAI(model="gpt-3.5-turbo-1106")

class Person(BaseModel):
    """A person's name and age."""

    name: str = Field(..., description="The person's name")
```

```

age: int = Field(..., description="The person's age")

chain = create_extraction_chain_pydantic(Person, model)
chain.invoke({'input':'Bob is 25 years old. He lives in New York.
He likes to play basketball. Sarah is 30 years old. She lives in San
Francisco. She likes to play tennis.''})

```

Output:

```
[Person(name='Bob', age=25), Person(name='Sarah', age=30)]
```

The Person Pydantic model has two properties, name and age; by calling the create\_extraction\_chain\_pydantic function with the input text, the LLM invokes the same function twice and creates two People objects.

## Query Planning

You may experience problems when user queries have multiple intents with intricate dependencies. *Query planning* is an effective way to parse a user's query into a series of steps that can be executed as a query graph with relevant dependencies:

```

from langchain_openai.chat_models import ChatOpenAI
from langchain.output_parsers.pydantic import PydanticOutputParser
from langchain_core.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
)
from pydantic.v1 import BaseModel, Field
from typing import List

class Query(BaseModel):
    id: int
    question: str
    dependencies: List[int] = Field(
        default_factory=list,
        description="""A list of sub-queries that must be completed before
        this task can be completed.
        Use a sub query when anything is unknown and we might need to ask
        many queries to get an answer.
        Dependencies must only be other queries."""
    )

class QueryPlan(BaseModel):
    query_graph: List[Query]

```

Defining QueryPlan and Query allows you to first ask an LLM to parse a user's query into multiple steps. Let's investigate how to create the query plan.

Input:

```
# Set up a chat model:  
model = ChatOpenAI()  
  
# Set up a parser:  
parser = PydanticOutputParser(pydantic_object=QueryPlan)  
  
template = """Generate a query plan. This will be used for task execution.  
  
Answer the following query: {query}  
  
Return the following query graph format:  
{format_instructions}  
"""  
system_message_prompt = SystemMessagePromptTemplate.from_template(template)  
chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt])  
  
# Create the LCEL chain with the prompt, model, and parser:  
chain = chat_prompt | model | parser  
  
result = chain.invoke({  
    "query": "I want to get the results from my database. Then I want to find  
    out what the average age of my top 10 customers is. Once I have the average  
    age, I want to send an email to John. Also I just generally want to send a  
    welcome introduction email to Sarah, regardless of the other tasks.",  
    "format_instructions": parser.get_format_instructions()})  
  
print(result.query_graph)
```

Output:

```
[Query(id=1, question='Get top 10 customers', dependencies=[]),  
 Query(id=2, question='Calculate average age of customers', dependencies=[1]),  
 Query(id=3, question='Send email to John', dependencies=[2]),  
 Query(id=4, question='Send welcome email to Sarah', dependencies=[])]
```

Initiate a ChatOpenAI instance and create a PydanticOutputParser for the QueryPlan structure. Then the LLM response is called and parsed, producing a structured query\_graph for your tasks with their unique dependencies.

## Creating Few-Shot Prompt Templates

Working with the generative capabilities of LLMs often involves making a choice between *zero-shot* and *few-shot learning (k-shot)*. While zero-shot learning requires no explicit examples and adapts to tasks based solely on the prompt, its dependence on the pretraining phase means it may not always yield precise results.

On the other hand, with few-shot learning, which involves providing a few examples of the desired task performance in the prompt, you have the opportunity to optimize the model's behavior, leading to more desirable outputs.

Due to the token LLM context length, you will often find yourself competing between adding lots of high-quality k-shot examples into your prompts while still aiming to generate an effective and deterministic LLM output.



Even as the token context window limit within LLMs continues to increase, providing a specific number of k-shot examples helps you minimize API costs.

Let's explore two methods for adding k-shot examples into your prompts with *few-shot prompt templates*: using *fixed examples* and using an *example selector*.

## Fixed-Length Few-Shot Examples

First, let's look at how to create a few-shot prompt template using a fixed number of examples. The foundation of this method lies in creating a robust set of few-shot examples:

```
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.prompts import (
    FewShotChatMessagePromptTemplate,
    ChatPromptTemplate,
)
examples = [
    {
        "question": "What is the capital of France?",
        "answer": "Paris",
    },
    {
        "question": "What is the capital of Spain?",
        "answer": "Madrid",
    } # ...more examples...
]
```

Each example is a dictionary containing a `question` and `answer` key that will be used to create pairs of `HumanMessage` and `AIMessage` messages.

## Formatting the Examples

Next, you'll configure a `ChatPromptTemplate` for formatting the individual examples, which will then be inserted into a `FewShotChatMessagePromptTemplate`.

Input:

```
example_prompt = ChatPromptTemplate.from_messages(  
    [  
        ("human", "{question}"),  
        ("ai", "{answer}")  
    ]  
)  
  
few_shot_prompt = FewShotChatMessagePromptTemplate(  
    example_prompt=example_prompt,  
    examples=examples,  
)  
  
print(few_shot_prompt.format())
```

Output:

```
Human: What is the capital of France?  
AI: Paris  
Human: What is the capital of Spain?  
AI: Madrid  
...more examples...
```

Notice how `example_prompt` will create `HumanMessage` and `AIMessage` pairs with the prompt inputs of `{question}` and `{answer}`.

After running `few_shot_prompt.format()`, the few-shot examples are printed as a string. As you'd like to use these within a `ChatOpenAI()` LLM request, let's create a new `ChatPromptTemplate`.

Input:

```
from langchain_core.output_parsers import StrOutputParser  
  
final_prompt = ChatPromptTemplate.from_messages(  
    [("system", '''You are responsible for answering  
    questions about countries. Only return the country  
    name.''),  
     few_shot_prompt, ("human", "{question}")],  
)  
  
model = ChatOpenAI()  
  
# Creating the LCEL chain with the prompt, model, and a StrOutputParser():  
chain = final_prompt | model | StrOutputParser()  
  
result = chain.invoke({"question": "What is the capital of America?"})  
  
print(result)
```

Output:

```
Washington, D.C.
```

After invoking the LCEL chain on `final_prompt`, your few-shot examples are added after the `SystemMessage`.

Notice that the LLM only returns 'Washington, D.C.' This is because after the LLMs response is returned, *it is parsed* by `StrOutputParser()`, an output parser. Adding `StrOutputParser()` is a common way to ensure that LLM responses in chains *return string values*. You'll explore this more in depth while learning sequential chains in LCEL.

## Selecting Few-Shot Examples by Length

Before diving into the code, let's outline your task. Imagine you're building a story-telling application powered by GPT-4. A user enters a list of character names with previously generated stories. However, each user's list of characters might have a different length. Including too many characters might generate a story that surpasses the LLM's context window limit. That's where you can use `LengthBasedExampleSelector` to adapt the prompt according to the length of user input:

```
from langchain_core.prompts import FewShotPromptTemplate, PromptTemplate
from langchain.prompts.example_selector import LengthBasedExampleSelector
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.messages import SystemMessage
import tiktoken

examples = [
    {"input": "Gollum", "output": "<Story involving Gollum>"},
    {"input": "Gandalf", "output": "<Story involving Gandalf>"},
    {"input": "Bilbo", "output": "<Story involving Bilbo>"},
]

story_prompt = PromptTemplate(
    input_variables=["input", "output"],
    template="Character: {input}\nStory: {output}",
)

def num_tokens_from_string(string: str) -> int:
    """Returns the number of tokens in a text string."""
    encoding = tiktoken.get_encoding("cl100k_base")
    num_tokens = len(encoding.encode(string))
    return num_tokens

example_selector = LengthBasedExampleSelector(
    examples=examples,
    example_prompt=story_prompt,
    max_length=1000, # 1000 tokens are to be included from examples
    # get_text_length: Callable[[str], int] = lambda x: len(re.split("\n", x))
    # You have modified the get_text_length function to work with the
    # TikToken library based on token usage:
    get_text_length=num_tokens_from_string,
)
```

First, you set up a `PromptTemplate` that takes two input variables for each example. Then `LengthBasedExampleSelector` adjusts the number of examples according to the *length of the examples input*, ensuring your LLM doesn't generate a story beyond its context window.

Also, you've customized the `get_text_length` function to use the `num_tokens_from_string` function that counts the total number of tokens using `tiktoken`. This means that `max_length=1000` represents the *number of tokens* rather than using the following default function:

```
get_text_length: Callable[[str], int] = lambda x: len(re.split("\n| ", x))
```

Now, to tie all these elements together:

```
dynamic_prompt = FewShotPromptTemplate(  
    example_selector=example_selector,  
    example_prompt=story_prompt,  
    prefix="Generate a story for {character} using the  
    current Character/Story pairs from all of the characters  
    as context.",  
    suffix="Character: {character}\nStory:",  
    input_variables=["character"],  
)  
  
# Provide a new character from Lord of the Rings:  
formatted_prompt = dynamic_prompt.format(character="Frodo")  
  
# Creating the chat model:  
chat = ChatOpenAI()  
  
response = chat.invoke([SystemMessage(content=formatted_prompt)])  
print(response.content)
```

Output:

```
Frodo was a young hobbit living a peaceful life in the Shire. However,  
his life...
```



### Provide Examples and Specify Format

When working with few-shot examples, the length of the content matters in determining how many examples the AI model can take into account. Tune the length of your input content and provide apt examples for efficient results to prevent the LLM from generating content that might surpass its context window limit.

After formatting the prompt, you create a chat model with `ChatOpenAI()` and load the formatted prompt into a `SystemMessage` that creates a small story about Frodo from *Lord of the Rings*.

Rather than creating and formatting a `ChatPromptTemplate`, it's often much easier to simply invoke a `SystemMessage` with a formatted prompt:

```
result = model.invoke([SystemMessage(content=formatted_prompt)])
```

## Limitations with Few-Shot Examples

Few-shot learning has limitations. Although it can prove beneficial in certain scenarios, it might not always yield the expected high-quality results. This is primarily due to two reasons:

- Pretrained models like GPT-4 can sometimes overfit to the few-shot examples, making them prioritize the examples over the actual prompt.
- LLMs have a token limit. As a result, there will always be a trade-off between the number of examples and the length of the response. Providing more examples might limit the response length and vice versa.

These limitations can be addressed in several ways. First, if few-shot prompting is not yielding the desired results, consider using differently framed phrases or experimenting with the language of the prompts themselves. Variations in how the prompt is phrased can result in different responses, highlighting the trial-and-error nature of prompt engineering.

Second, think about including explicit instructions to the model to ignore the examples after it understands the task or to use the examples just for formatting guidance. This might influence the model to not overfit to the examples.

If the tasks are complex and the performance of the model with few-shot learning is not satisfactory, you might need to consider [fine-tuning](#) your model. Fine-tuning provides a more nuanced understanding of a specific task to the model, thus improving the performance significantly.

## Saving and Loading LLM Prompts

To effectively leverage generative AI models such as GPT-4, it is beneficial to store prompts as files instead of Python code. This approach enhances the shareability, storage, and versioning of your prompts.

LangChain supports both saving and loading prompts from JSON and YAML. Another key feature of LangChain is its support for detailed specification in one file or distributed across multiple files. This means you have the flexibility to store different components such as templates, examples, and others in distinct files and reference them as required.

Let's learn how to save and load prompts:

```
from langchain_core.prompts import PromptTemplate, load_prompt

prompt = PromptTemplate(
    template='''Translate this sentence from English to Spanish.
\nSentence: {sentence}\nTranslation:''',
    input_variables=["sentence"],
)

prompt.save("translation_prompt.json")

# Loading the prompt template:
load_prompt("translation_prompt.json")
# Returns PromptTemplate()
```

After importing `PromptTemplate` and `load_prompt` from the `langchain.prompts` module, you define a `PromptTemplate` for English-to-Spanish translation tasks and save it as `translation_prompt.json`. Finally, you load the saved prompt template using the `load_prompt` function, which returns an instance of `PromptTemplate`.



Please be aware that LangChain's prompt saving may not work with all types of prompt templates. To mitigate this, you can utilize the `pickle` library or `.txt` files to read and write any prompts that LangChain does not support.

You've learned how to create few-shot prompt templates using LangChain with two techniques: a fixed number of examples and using an example selector.

The former creates a set of few-shot examples and uses a `ChatPromptTemplate` object to format these into chat messages. This forms the basis for creating a `FewShotChatMessagePromptTemplate` object.

The latter approach, using an example selector, is handy when user input varies significantly in length. In such scenarios, a `LengthBasedExampleSelector` can be utilized to adjust the number of examples based on user input length. This ensures your LLM does not exceed its context window limit.

Moreover, you've seen how easy it is to store/load prompts as files, enabling enhanced shareability, storage, and versioning.

# Data Connection

Harnessing an LLM application, coupled with your data, uncovers a plethora of opportunities to boost efficiency while refining your decision-making processes.

Your organization's data may manifest in various forms:

## Unstructured data

This could include Google Docs, threads from communication platforms such as Slack or Microsoft Teams, web pages, internal documentation, or code repositories on GitHub.

## Structured data

Data neatly housed within SQL, NoSQL, or Graph databases.

To query your unstructured data, a process of loading, transforming, embedding, and subsequently storing it within a vector database is necessary. A *vector database* is a specialized type of database designed to efficiently store and query data in the form of vectors, which represent complex data like text or images in a format suitable for machine learning and similarity search.

As for structured data, given its already indexed and stored state, you can utilize a LangChain agent to conduct an intermediate query on your database. This allows for the extraction of specific features, which can then be used within your LLM prompts.

There are multiple Python packages that can help with your data ingestion, including [Unstructured](#), [LlamaIndex](#), and [LangChain](#).

**Figure 4-2** illustrates a standardized approach to data ingestion. It begins with the data sources, which are then loaded into documents. These documents are then chunked and stored within a vector database for later retrieval.

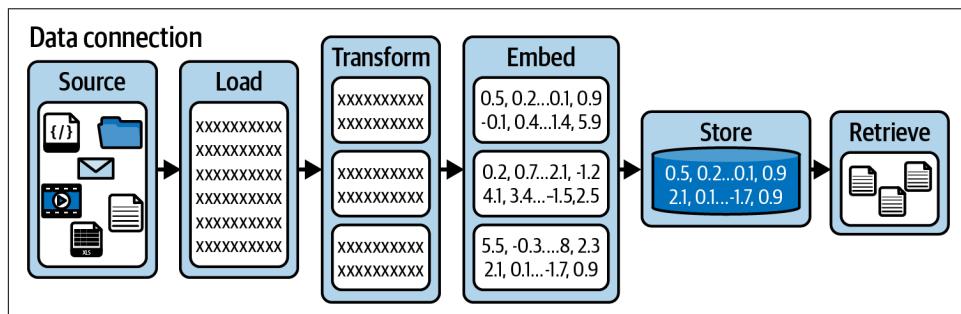


Figure 4-2. A data connection to retrieval pipeline

In particular LangChain equips you with essential components to load, modify, store, and retrieve your data:

#### *Document loaders*

These facilitate uploading informational resources, or *documents*, from a diverse range of sources such as Word documents, PDF files, text files, or even web pages.

#### *Document transformers*

These tools allow the segmentation of documents, conversion into a Q&A layout, elimination of superfluous documents, and much more.

#### *Text embedding models*

These can transform unstructured text into a sequence of floating-point numbers used for similarity search by vector stores.

#### *Vector databases (vector stores)*

These databases can save and execute searches over embedded data.

#### *Retrievers*

These tools offer the capability to query and retrieve data.

Also, it's worth mentioning that other LLM frameworks such as [LlamaIndex](#) work seamlessly with LangChain. [LlamaHub](#) is another open source library dedicated to document loaders and can create LangChain-specific Document objects.

## Document Loaders

Let's imagine you've been tasked with building an LLM data collection pipeline for NutriFusion Foods. The information that you need to gather for the LLM is contained within:

- A PDF of a book called *Principles of Marketing*
- Two *.docx* marketing reports in a public Google Cloud Storage bucket
- Three *.csv* files showcasing the marketing performance data for 2021, 2022, and 2023

Create a new Jupyter Notebook or Python file in *content/chapter\_4* of the [shared repository](#), and then run `pip install pdf2image docx2txt pypdf`, which will install three packages.

All of the data apart from `.docx` files can be found in `content/chapter_4/data`. You can start by importing all of your various data loaders and creating an empty `all_documents` list to store all of the `Document` objects across your data sources.

Input:

```
from langchain_community.document_loaders import Docx2txtLoader
from langchain_community.document_loaders import PyPDFLoader
from langchain_community.document_loaders.csv_loader import CSVLoader
import glob
from langchain.text_splitter import CharacterTextSplitter

# To store the documents across all data sources:
all_documents = []

# Load the PDF:
loader = PyPDFLoader("data/principles_of_marketing_book.pdf")
pages = loader.load_and_split()
print(pages[0])

# Add extra metadata to each page:
for page in pages:
    page.metadata["description"] = "Principles of Marketing Book"

# Checking that the metadata has been added:
for page in pages[0:2]:
    print(page.metadata)

# Saving the marketing book pages:
all_documents.extend(pages)

csv_files = glob.glob("data/*.csv")

# Filter to only include the word Marketing in the file name:
csv_files = [f for f in csv_files if "Marketing" in f]

# For each .csv file:
for csv_file in csv_files:
    loader = CSVLoader(file_path=csv_file)
    data = loader.load()
    # Saving the data to the all_documents list:
    all_documents.extend(data)

text_splitter = CharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=200, chunk_overlap=0
)

urls = [
    '''https://storage.googleapis.com/oreilly-content/NutriFusion%20Foods%20Marketing%20Plan%202022.docx''',
    '''https://storage.googleapis.com/oreilly-content/NutriFusion%20Foods%20Plan%202022.xlsx'''
]
```

```

    @Marketing%20Plan%202023.docx''' ,
]

docs = []
for url in urls:
    loader = Docx2txtLoader(url.replace('\n', ' '))
    pages = loader.load()
    chunks = text_splitter.split_documents(pages)

    # Adding the metadata to each chunk:
    for chunk in chunks:
        chunk.metadata["source"] = "NutriFusion Foods Marketing Plan - 2022/2023"
    docs.extend(chunks)

# Saving the marketing book pages:
all_documents.extend(docs)

```

Output:

```

page_content='Principles of Marketing'
metadata={'source': 'data/principles_of_marketing_book.pdf', 'page': 0}
{'source': 'data/principles_of_marketing_book.pdf', 'page': 0,
'description': 'Principles of Marketing Book'}
{'source': 'data/principles_of_marketing_book.pdf', 'page': 1,
'description': 'Principles of Marketing Book'}

```

Then using PyPDFLoader, you can import a `.pdf` file and split it into multiple pages using the `.load_and_split()` function.

Additionally, it's possible to add extra metadata to each page because the metadata is a Python dictionary on each Document object. Also, notice in the preceding output for Document objects the metadata source is attached to.

Using the package `glob`, you can easily find all of the `.csv` files and individually load these into LangChain Document objects with a CSVLoader.

Finally, the two marketing reports are loaded from a public Google Cloud Storage bucket and are then split into 200 token-chunk sizes using a `text_splitter`.

This section equipped you with the necessary knowledge to create a comprehensive document-loading pipeline for NutriFusion Foods' LLM. Starting with data extraction from a PDF, several CSV files and two `.docx` files, each document was enriched with relevant metadata for better context.

You now have the ability to seamlessly integrate data from a variety of document sources into a cohesive data pipeline.

# Text Splitters

Balancing the length of each document is also a crucial factor. If a document is too lengthy, it may surpass the *context length* of the LLM (the maximum number of tokens that an LLM can process within a single request). But if the documents are excessively fragmented into smaller chunks, there's a risk of losing significant contextual information, which is equally undesirable.

You might encounter specific challenges while text splitting, such as:

- Special characters such as hashtags, @ symbols, or links might not split as anticipated, affecting the overall structure of the split documents.
- If your document contains intricate formatting like tables, lists, or multilevel headings, the text splitter might find it difficult to retain the original formatting.

There are ways to overcome these challenges that we'll explore later.

This section introduces you to text splitters in LangChain, tools utilized to break down large chunks of text to better adapt to your model's context window.



There isn't a perfect document size. Start by using good heuristics and then build a training/test set that you can use for LLM evaluation.

LangChain provides a range of text splitters so that you can easily split by any of the following:

- Token count
- Recursively by multiple characters
- Character count
- Code
- Markdown headers

Let's explore three popular splitters: `CharacterTextSplitter`, `TokenTextSplitter`, and `RecursiveCharacterTextSplitter`.

## Text Splitting by Length and Token Size

In [Chapter 3](#), you learned how to count the number of tokens within a GPT-4 call with `tiktoken`. You can also use `tiktoken` to split strings into appropriately sized chunks and documents.

Remember to install tiktoken and langchain-text-splitters with `pip install tiktoken langchain-text-splitters`.

To split by token count in LangChain, you can use a `CharacterTextSplitter` with a `.from_tiktoken_encoder()` function.

You'll initially create a `CharacterTextSplitter` with a chunk size of 50 characters and no overlap. Using the `split_text` method, you're chopping the text into pieces and then printing out the total number of chunks created.

Then you'll do the same thing, but this time with a *chunk overlap* of 48 characters. This shows how the number of chunks changes based *on whether you allow overlap*, illustrating the impact of these settings on how your text gets divided:

```
from langchain_text_splitters import CharacterTextSplitter

text = """
Biology is a fascinating and diverse field of science that explores the
living world and its intricacies \n\n. It encompasses the study of life, its
origins, diversity, structure, function, and interactions at various levels
from molecules and cells to organisms and ecosystems \n\n. In this 1000-word
essay, we will delve into the core concepts of biology, its history, key
areas of study, and its significance in shaping our understanding of the
natural world. \n\n ... (truncated to save space)...
"""

# No chunk overlap:
text_splitter = CharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=50, chunk_overlap=0, separator="\n",
)
texts = text_splitter.split_text(text)
print(f"Number of texts with no chunk overlap: {len(texts)}")

# Including a chunk overlap:
text_splitter = CharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=50, chunk_overlap=48, separator="\n",
)
texts = text_splitter.split_text(text)
print(f"Number of texts with chunk overlap: {len(texts)})
```

Output:

```
Number of texts with no chunk overlap: 3
Number of texts with chunk overlap: 6
```

In the previous section, you used the following to load and split the `.pdf` into LangChain documents:

```
pages = loader.load_and_split()
```

It's possible for you to have more granular control on the size of each document by creating a `TextSplitter` and attaching it to your `Document` loading pipelines:

```
def load_and_split(text_splitter: TextSplitter | None = None) ->
List[Document]
```

Simply create a `TokenTextSplitter` with a `chunk_size=500` and a `chunk_overlap` of 50:

```
from langchain.text_splitter import TokenTextSplitter
from langchain_community.document_loaders import PyPDFLoader

text_splitter = TokenTextSplitter(chunk_size=500, chunk_overlap=50)
loader = PyPDFLoader("data/principles_of_marketing_book.pdf")
pages = loader.load_and_split(text_splitter=text_splitter)

print(len(pages)) #737
```

The *Principles of Marketing* book contains 497 pages, but after using a `TokenTextSplitter` with a `chunk_size` of 500 tokens, you've created 776 smaller LangChain Document objects.

## Text Splitting with Recursive Character Splitting

Dealing with sizable blocks of text can present unique challenges in text analysis. A helpful strategy for such situations involves the use of *recursive character splitting*. This method facilitates the division of a large body of text into manageable segments, making further analysis more accessible.

This approach becomes incredibly effective when handling generic text. It leverages a list of characters as parameters and sequentially splits the text based on these characters. The resulting sections continue to be divided until they reach an acceptable size. By default, the character list comprises "\n\n", "\n", " ", and "". This arrangement aims to retain the integrity of paragraphs, sentences, and words, preserving the semantic context.

The process hinges on the character list provided and sizes the resulting sections based on the character count.

Before diving into the code, it's essential to understand what the `RecursiveCharacterTextSplitter` does. It takes a text and a list of delimiters (characters that define the boundaries for splitting the text). Starting from the first delimiter in the list, the splitter attempts to divide the text. If the resulting chunks are still too large, it proceeds to the next delimiter, and so on. This process continues *recursively* until the chunks are small enough or all delimiters are exhausted.

Using the preceding `text` variable, start by importing `RecursiveCharacterTextSplitter`. This instance will be responsible for splitting the text. When initializing

the splitter, parameters `chunk_size`, `chunk_overlap`, and `length_function` are set. Here, `chunk_size` is set to 100, and `chunk_overlap` to 20.

The `length_function` is defined as `len` to determine the size of the chunks. It's also possible to modify the `length_function` argument to use a tokenizer count instead of using the default `len` function, which will count characters:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=100,
    chunk_overlap=20,
    length_function=len,
)
```

Once the `text_splitter` instance is ready, you can use `.split_text` to split the `text` variable into smaller chunks. These chunks are stored in the `texts` Python list:

```
# Split the text into chunks:
texts = text_splitter.split_text(text)
```

As well as simply splitting the text with overlap into a list of strings, you can easily create LangChain Document objects with the `.create_documents` function. Creating Document objects is useful because it allows you to:

- Store documents within a vector database for semantic search
- Add metadata to specific pieces of text
- Iterate over multiple documents to create a higher-level summary

To add metadata, provide a list of dictionaries to the `metadatas` argument:

```
# Create documents from the chunks:
metadatas = {"title": "Biology", "author": "John Doe"}
docs = text_splitter.create_documents(texts, metadatas=[metadatas] * len(texts))
```

But what if your existing Document objects are too long?

You can easily handle that by using the `.split_documents` function with a `TextSplitter`. This will take in a list of Document objects and will return a new list of Document objects based on your `TextSplitter` class argument settings:

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=300)
splitted_docs = text_splitter.split_documents(docs)
```

You've now gained the ability to craft an efficient data loading pipeline, leveraging sources such as PDFs, CSVs, and Google Cloud Storage links. Furthermore, you've learned how to enrich the collected documents with relevant metadata, providing meaningful context for analysis and prompt engineering.

With the introduction of text splitters, you can now strategically manage document sizes, optimizing for both the LLM's context window and the preservation of context-rich information. You've navigated handling larger texts by employing recursion and character splitting. This newfound knowledge empowers you to work seamlessly with various document sources and integrate them into a robust data pipeline.

## Task Decomposition

*Task decomposition* is the strategic process of dissecting complex problems into a suite of manageable subproblems. This approach aligns seamlessly with the natural tendencies of software engineers, who often conceptualize tasks as interrelated sub-components.

In software engineering, by utilizing task decomposition you can reduce cognitive burden and harness the advantages of problem isolation and adherence to the single responsibility principle.

Interestingly, LLMs stand to gain considerably from the application of task decomposition across a range of use cases. This approach aids in maximizing the utility and effectiveness of LLMs in problem-solving scenarios by enabling them to handle intricate tasks that would be challenging to resolve as a single entity, as illustrated in [Figure 4-3](#).

Here are several examples of LLMs using decomposition:

### *Complex problem solving*

In instances where a problem is multifaceted and cannot be solved through a single prompt, task decomposition is extremely useful. For example, solving a complex legal case could be broken down into understanding the case's context, identifying relevant laws, determining legal precedents, and crafting arguments. Each subtask can be solved independently by an LLM, providing a comprehensive solution when combined.

### *Content generation*

For generating long-form content such as articles or blogs, the task can be decomposed into generating an outline, writing individual sections, and then compiling and refining the final draft. Each step can be individually managed by GPT-4 for better results.

### *Large document summary*

Summarizing lengthy documents such as research papers or reports can be done more effectively by decomposing the task into several smaller tasks, like understanding individual sections, summarizing them independently, and then compiling a final summary.

### *Interactive conversational agents*

For creating advanced chatbots, task decomposition can help manage different aspects of conversation such as understanding user input, maintaining context, generating relevant responses, and managing dialogue flow.

### *Learning and tutoring systems*

In digital tutoring systems, decomposing the task of teaching a concept into understanding the learner's current knowledge, identifying gaps, suggesting learning materials, and evaluating progress can make the system more effective. Each subtask can leverage GPT-4's generative abilities.

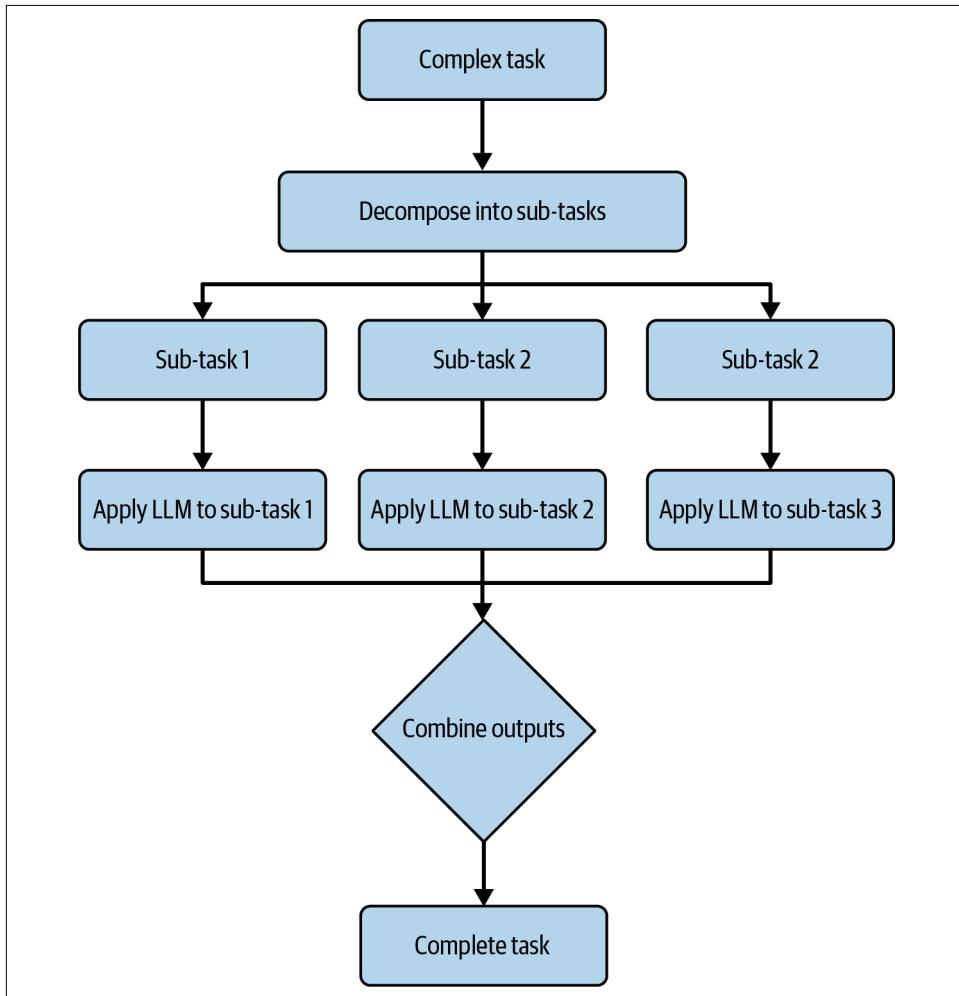


Figure 4-3. Task decomposition with LLMs



## Divide Labor

Task decomposition is a crucial strategy for you to tap into the full potential of LLMs. By dissecting complex problems into simpler, manageable tasks, you can leverage the problem-solving abilities of these models more effectively and efficiently.

In the sections ahead, you'll learn how to create and integrate multiple LLM chains to orchestrate more complicated workflows.

## Prompt Chaining

Often you'll find that attempting to do a single task within one prompt is impossible. You can utilize a mixture of *prompt chaining*, which involves combining multiple prompt inputs/outputs with specifically tailored LLM prompts to build up an idea.

Let's imagine an example with a film company that would like to partially automate their film creation. This could be broken down into several key components, such as:

- Character creation
- Plot generation
- Scenes/world building

Figure 4-4 shows what the prompt workflow might look like.

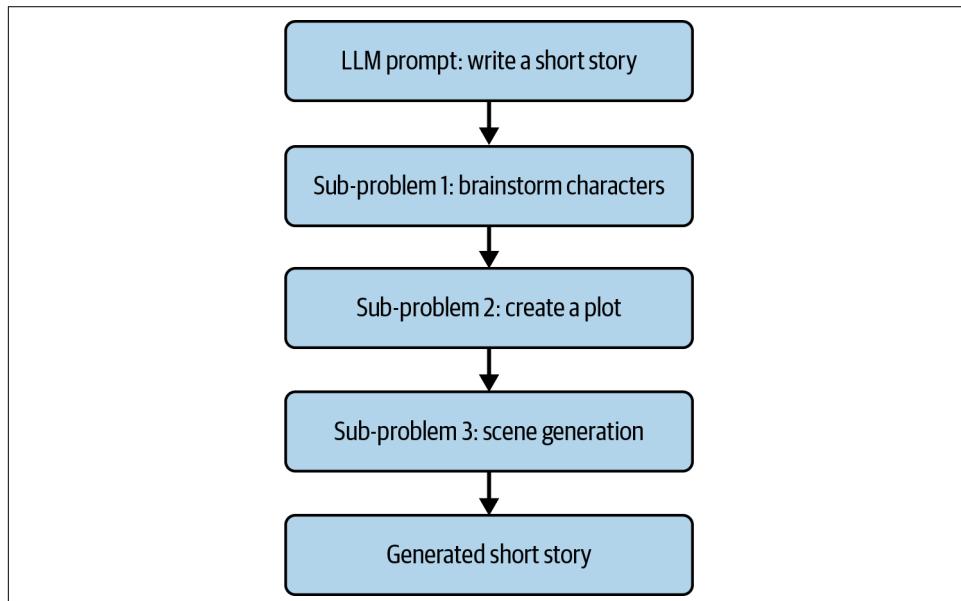


Figure 4-4. A sequential story creation process

## Sequential Chain

Let's decompose the task into *multiple chains* and recompose them into a single chain:

`character_generation_chain`

A chain responsible for creating multiple characters given a 'genre'.

`plot_generation_chain`

A chain that will create the plot given the 'characters' and 'genre' keys.

`scene_generation_chain`

This chain will generate any missing scenes that were not initially generated from the `plot_generation_chain`.

Let's start by creating three separate `ChatPromptTemplate` variables, one for each chain:

```
from langchain_core.prompts.chat import ChatPromptTemplate

character_generation_prompt = ChatPromptTemplate.from_template(
    """I want you to brainstorm three to five characters for my short story. The
    genre is {genre}. Each character must have a Name and a Biography.
    You must provide a name and biography for each character, this is very
    important!
    ---
    Example response:
    Name: CharWiz, Biography: A wizard who is a master of magic.
    Name: CharWar, Biography: A warrior who is a master of the sword.
    ---
    Characters: """
)

plot_generation_prompt = ChatPromptTemplate.from_template(
    """Given the following characters and the genre, create an effective
    plot for a short story:
    Characters:
    {characters}
    ---
    Genre: {genre}
    ---
    Plot: """
)

scene_generation_plot_prompt = ChatPromptTemplate.from_template(
    """Act as an effective content creator.
    Given multiple characters and a plot, you are responsible for
    generating the various scenes for each act.

    You must decompose the plot into multiple effective scenes:
    ---
    Characters:
    """
)
```

```

{characters}
---
Genre: {genre}
---
Plot: {plot}
---
Example response:
Scenes:
Scene 1: Some text here.
Scene 2: Some text here.
Scene 3: Some text here.
-----
Scenes:
"""
)

```

Notice that as the prompt templates flow from character to plot and scene generation, you add more placeholder variables from the previous steps.

The question remains, how can you guarantee that these extra strings are available for your downstream ChatPromptTemplate variables?

## itemgetter and Dictionary Key Extraction

Within LCEL you can use the `itemgetter` function from the `operator` package to extract keys from the previous step, as long as a dictionary was present within the previous step:

```

from operator import itemgetter
from langchain_core.runnables import RunnablePassthrough

chain = RunnablePassthrough() | {
    "genre": itemgetter("genre"),
}
chain.invoke({"genre": "fantasy"})
# {'genre': 'fantasy'}

```

The `RunnablePassThrough` function simply passes any inputs directly to the next step. Then a new dictionary is created by using the same key within the `invoke` function; this key is extracted by using `itemgetter("genre")`.

It's essential to use the `itemgetter` function throughout parts of your LCEL chains so that any subsequent `ChatPromptTemplate` placeholder variables will always have valid values.

Additionally, you can use `lambda` or `RunnableLambda` functions within an LCEL chain to manipulate previous dictionary values. A `lambda` is an anonymous function within Python:

```
from langchain_core.runnables import RunnableLambda
```

```

chain = RunnablePassthrough() | {
    "genre": itemgetter("genre"),
    "upper_case_genre": lambda x: x["genre"].upper(),
    "lower_case_genre": RunnableLambda(lambda x: x["genre"].lower()),
}
chain.invoke({"genre": "fantasy"})
# {'genre': 'fantasy', 'upper_case_genre': 'FANTASY',
# 'lower_case_genre': 'fantasy'}

```

Now that you're aware of how to use `RunnablePassThrough`, `itemgetter`, and `lambda` functions, let's introduce one final piece of syntax: `RunnableParallel`:

```

from langchain_core.runnables import RunnableParallel

master_chain = RunnablePassthrough() | {
    "genre": itemgetter("genre"),
    "upper_case_genre": lambda x: x["genre"].upper(),
    "lower_case_genre": RunnableLambda(lambda x: x["genre"].lower()),
}

master_chain_two = RunnablePassthrough() | RunnableParallel(
    genre=itemgetter("genre"),
    upper_case_genre=lambda x: x["genre"].upper(),
    lower_case_genre=RunnableLambda(lambda x: x["genre"].lower()),
)

story_result = master_chain.invoke({"genre": "Fantasy"})
print(story_result)

story_result = master_chain_two.invoke({"genre": "Fantasy"})
print(story_result)

# master chain: {'genre': 'Fantasy', 'upper_case_genre': 'FANTASY',
# 'lower_case_genre': 'fantasy'}
# master chain two: {'genre': 'Fantasy', 'upper_case_genre': 'FANTASY',
# 'lower_case_genre': 'fantasy'}

```

First, you import `RunnableParallel` and create two LCEL chains called `master_chain` and `master_chain_two`. These are then invoked with exactly the same arguments; the `RunnablePassthrough` then passes the dictionary into the second part of the chain.

The second part of `master_chain` and `master_chain_two` will return exactly the *same result*.

So rather than directly using a dictionary, you can choose to use a `RunnableParallel` function instead. These two chain outputs *are interchangeable*, so choose whichever syntax you find more comfortable.

Let's create three LCEL chains using the prompt templates:

```
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser

# Create the chat model:
model = ChatOpenAI()

# Create the subchains:
character_generation_chain = ( character_generation_prompt
| model
| StrOutputParser() )

plot_generation_chain = ( plot_generation_prompt
| model
| StrOutputParser() )

scene_generation_plot_chain = ( scene_generation_plot_prompt
| model
| StrOutputParser() )
```

After creating all the chains, you can then attach them to a master LCEL chain.

Input:

```
from langchain_core.runnables import RunnableParallel
from operator import itemgetter
from langchain_core.runnables import RunnablePassthrough

master_chain = (
    {"characters": character_generation_chain, "genre": RunnablePassthrough()
     | RunnableParallel(
        characters=itemgetter("characters"),
        genre=itemgetter("genre"),
        plot=plot_generation_chain,
    )
    | RunnableParallel(
        characters=itemgetter("characters"),
        genre=itemgetter("genre"),
        plot=itemgetter("plot"),
        scenes=scene_generation_plot_chain,
    )
)
story_result = master_chain.invoke({"genre": "Fantasy"})
```

The output is truncated when you see ... to save space. However, in total there were five characters and nine scenes generated.

## Output:

```
{'characters': '''Name: Lyra, Biography: Lyra is a young elf who possesses ..\n\nName: Orion, Biography: Orion is a ..''', 'genre': {'genre': 'Fantasy'} } 'plot': '''In the enchanted forests of a mystical realm, a great darkness looms, threatening to engulf the land and its inhabitants. Lyra, the young elf with a deep connection to nature, ...''', 'scenes': '''Scene 1: Lyra senses the impending danger in the forest ...\\n\\nScene 2: Orion, on his mission to investigate the disturbances in the forest...\\n\\nScene 9: After the battle, Lyra, Orion, Seraphina, Finnegan...'''}
```

The scenes are split into separate items within a Python list. Then two new prompts are created to generate both a character script and a summarization prompt:

```
# Extracting the scenes using .split('\\n') and removing empty strings:
scenes = [scene for scene in story_result["scenes"].split("\\n") if scene]
generated_scenes = []
previous_scene_summary = ""

character_script_prompt = ChatPromptTemplate.from_template(
    template="""Given the following characters: {characters} and the genre: {genre}, create an effective character script for a scene.

    You must follow the following principles:
    - Use the Previous Scene Summary: {previous_scene_summary} to avoid
    repeating yourself.
    - Use the Plot: {plot} to create an effective scene character script.
    - Currently you are generating the character dialogue script for the
    following scene: {scene}

    ---
    Here is an example response:
    SCENE 1: ANNA'S APARTMENT

    (ANNA is sorting through old books when there is a knock at the door.
    She opens it to reveal JOHN.)
    ANNA: Can I help you, sir?
    JOHN: Perhaps, I think it's me who can help you. I heard you're
    researching time travel.
    (Anna looks intrigued but also cautious.)
    ANNA: That's right, but how do you know?
    JOHN: You could say... I'm a primary source.

    ---
    SCENE NUMBER: {index}

    """
)
summarize_prompt = ChatPromptTemplate.from_template(
    template="""Given a character script, create a summary of the scene.
    Character script: {character_script}""",
)
```

Technically, you could generate all of the scenes asynchronously. However, it's beneficial to know what each character has done in the *previous scene* to avoid repeating points.

Therefore, you can create two LCEL chains, one for generating the character scripts per scene and the other for summarizations of previous scenes:

```
# Loading a chat model:
model = ChatOpenAI(model='gpt-3.5-turbo-16k')

# Create the LCEL chains:
character_script_generation_chain = (
    {
        "characters": RunnablePassthrough(),
        "genre": RunnablePassthrough(),
        "previous_scene_summary": RunnablePassthrough(),
        "plot": RunnablePassthrough(),
        "scene": RunnablePassthrough(),
        "index": RunnablePassthrough(),
    }
    | character_script_prompt
    | model
    | StrOutputParser()
)

summarize_chain = summarize_prompt | model | StrOutputParser()

# You might want to use tqdm here to track the progress,
# or use all of the scenes:
for index, scene in enumerate(scenes[0:3]):

    # # Create a scene generation:
    scene_result = character_script_generation_chain.invoke(
        {
            "characters": story_result["characters"],
            "genre": "fantasy",
            "previous_scene_summary": previous_scene_summary,
            "index": index,
        }
    )

    # Store the generated scenes:
    generated_scenes.append(
        {"character_script": scene_result, "scene": scenes[index]}
    )

    # If this is the first scene then we don't have a
    # previous scene summary:
    if index == 0:
        previous_scene_summary = scene_result
    else:
        # If this is the second scene or greater then
```

```

# we can use and generate a summary:
summary_result = summarize_chain.invoke(
    {"character_script": scene_result}
)
previous_scene_summary = summary_result

```

First, you'll establish a `character_script_generation_chain` in your script, utilizing various runnables like `RunnablePassthrough` for smooth data flow. Crucially, this chain integrates model = `ChatOpenAI(model='gpt-3.5-turbo-16k')`, a powerful model with a generous 16k context window, ideal for extensive content generation tasks. When invoked, this chain adeptly generates character scripts, drawing on inputs such as character profiles, genre, and scene specifics.

You dynamically enrich each scene by adding the summary of the previous scene, creating a simple yet effective buffer memory. This technique ensures continuity and context in the narrative, enhancing the LLM's ability to generate coherent character scripts.

Additionally, you'll see how the `StrOutputParser` elegantly converts model outputs into structured strings, making the generated content easily usable.



### Divide Labor

Remember, designing your tasks in a sequential chain greatly benefits from the Divide Labor principle. Breaking tasks down into smaller, manageable chains can increase the overall quality of your output. Each chain in the sequential chain contributes its individual effort toward achieving the overarching task goal.

Using chains gives you the ability to use different models. For example, using a smart model for the ideation and a cheap model for the generation usually gives optimal results. This also means you can have fine-tuned models on each step.

## Structuring LCEL Chains

In LCEL you must ensure that the first part of your LCEL chain is a *Runnable* type. The following code will throw an error:

```

from langchain_core.prompts.chat import ChatPromptTemplate
from operator import itemgetter
from langchain_core.runnables import RunnablePassthrough, RunnableLambda

bad_first_input = {
    "film_required_age": 18,
}

prompt = ChatPromptTemplate.from_template(
    "Generate a film title, the age is {film_required_age}"
)

```

```
)  
  
# This will error:  
bad_chain = bad_first_input | prompt
```

A Python dictionary with a value of 18 will not create a runnable LCEL chain. However, all of the following implementations will work:

```
# All of these chains enforce the runnable interface:  
first_good_input = {"film_required_age": itemgetter("film_required_age")}  
  
# Creating a dictionary within a RunnableLambda:  
second_good_input = RunnableLambda(lambda x: { "film_required_age":  
x["film_required_age"] } )  
  
third_good_input = RunnablePassthrough()  
fourth_good_input = {"film_required_age": RunnablePassthrough()}  
# You can also create a chain starting with RunnableParallel(...)  
  
first_good_chain = first_good_input | prompt  
second_good_chain = second_good_input | prompt  
third_good_chain = third_good_input | prompt  
fourth_good_chain = fourth_good_input | prompt  
  
first_good_chain.invoke({  
    "film_required_age": 18  
}) # ...
```

Sequential chains are great at incrementally building generated knowledge that is used by future chains, but they often yield slower response times due to their sequential nature. As such, `SequentialChain` data pipelines are best suited for server-side tasks, where immediate responses are not a priority and users aren't awaiting real-time feedback.

## Document Chains

Let's imagine that before accepting your generated story, the local publisher has requested that you provide a summary based on all of the character scripts. This is a good use case for *document chains* because you need to provide an LLM with a large amount of text that wouldn't fit within a single LLM request due to the context length restrictions.

Before delving into the code, let's first get a sense of the broader picture. The script you are going to see performs a text summarization task on a collection of scenes.

Remember to install Pandas with `pip install pandas`.

Now, let's start with the first set of code:

```
from langchain_text_splitters import CharacterTextSplitter
from langchain.chains.summarize import load_summarize_chain
import pandas as pd
```

These lines are importing all the necessary tools you need. `CharacterTextSplitter` and `load_summarize_chain` are from the LangChain package and will help with text processing, while Pandas (imported as `pd`) will help manipulate your data.

Next, you'll be dealing with your data:

```
df = pd.DataFrame(generated_scenes)
```

Here, you create a Pandas DataFrame from the `generated_scenes` variable, effectively converting your raw scenes into a tabular data format that Pandas can easily manipulate.

Then you need to consolidate your text:

```
all_character_script_text = "\n".join(df.character_script.tolist())
```

In this line, you're transforming the `character_script` column from your DataFrame into a single text string. Each entry in the column is converted into a list item, and all items are joined together with new lines in between, resulting in a single string that contains all character scripts.

Once you have your text ready, you prepare it for the summarization process:

```
text_splitter = CharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=1500, chunk_overlap=200
)
docs = text_splitter.create_documents([all_character_script_text])
```

Here, you create a `CharacterTextSplitter` instance using its class method `from_tiktoken_encoder`, with specific parameters for chunk size and overlap. You then use this text splitter to split your consolidated script text into chunks suitable for processing by your summarization tool.

Next, you set up your summarization tool:

```
chain = load_summarize_chain(llm=model, chain_type="map_reduce")
```

This line is about setting up your summarization process. You're calling a function that loads a summarization chain with a chat model in a map-reduce style approach.

Then you run the summarization:

```
summary = chain.invoke(docs)
```

This is where you actually perform the text summarization. The `invoke` method executes the summarization on the chunks of text you prepared earlier and stores the summary into a variable.

Finally, you print the result:

```
print(summary['output_text'])
```

This is the culmination of all your hard work. The resulting summary text is printed to the console for you to see.

This script takes a collection of scenes, consolidates the text, chunks it up, summarizes it, and then prints the summary:

```
from langchain.text_splitter import CharacterTextSplitter
from langchain.chains.summarize import load_summarize_chain
import pandas as pd

df = pd.DataFrame(generated_scenes)

all_character_script_text = "\n".join(df.character_script.tolist())

text_splitter = CharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=1500, chunk_overlap=200
)

docs = text_splitter.create_documents([all_character_script_text])

chain = load_summarize_chain(llm=model, chain_type="map_reduce")
summary = chain.invoke(docs)
print(summary['output_text'])
```

Output:

```
Aurora and Magnus agree to retrieve a hidden artifact, and they enter an
ancient library to find a book that will guide them to the relic...'.
```

It's worth noting that even though you've used a `map_reduce` chain, there are four core chains for working with `Document` objects within LangChain.

## Stuff

The document insertion chain, also referred to as the *stuff* chain (drawing from the concept of *stuffing* or *filling*), is the simplest approach among various document chaining strategies. [Figure 4-5](#) illustrates the process of integrating multiple documents into a single LLM request.

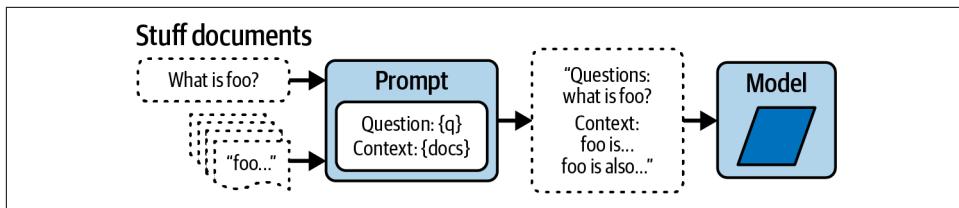


Figure 4-5. Stuff documents chain

## Refine

The refine documents chain (Figure 4-6) creates an LLM response through a cyclical process that *iteratively updates its output*. During each loop, it combines the current output (derived from the LLM) with the current document. Another LLM request is made to *update the current output*. This process continues until all documents have been processed.

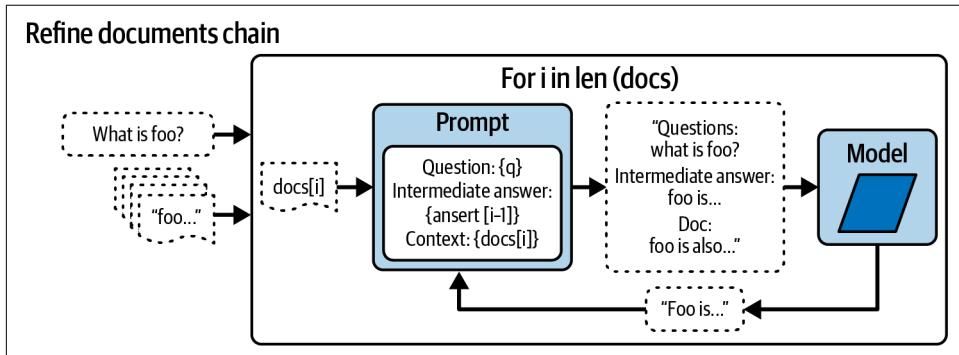


Figure 4-6. Refine documents chain

## Map Reduce

The map reduce documents chain in Figure 4-7 starts with an LLM chain to each separate document (a process known as the Map step), interpreting the resulting output as a newly generated document.

Subsequently, all these newly created documents are introduced to a distinct combine documents chain to formulate a singular output (a process referred to as the Reduce step). If necessary, to ensure the new documents seamlessly fit into the context length, an optional compression process is used on the mapped documents. If required, this compression happens recursively.

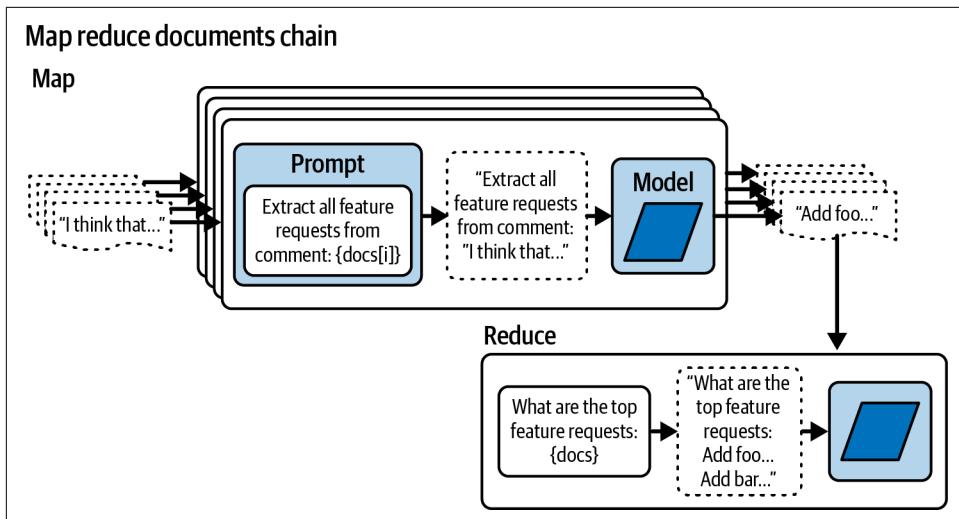


Figure 4-7. Map reduce documents chain

## Map Re-rank

There is also map re-rank, which operates by executing an initial prompt on each document. This not only strives to fulfill a given task but also assigns a confidence score reflecting the certainty of its answer. The response with the highest confidence score is then selected and returned.

**Table 4-1** demonstrates the advantages and disadvantages for choosing a specific document chain strategy.

Table 4-1. Overview of document chain strategies

| Approach                    | Advantages  | Disadvantages   |
|-----------------------------|---|---|
| Stuff Documents Chain       | Simple to implement. Ideal for scenarios with small documents and few inputs.   | May not be suitable for handling large documents or multiple inputs due to prompt size limitation.  |
| Refine Documents Chain      | Allows iterative refining of the response. More control over each step of response generation. Good for progressive extraction tasks. | Might not be optimal for real-time applications due to the loop process.  |
| Map Reduce Documents Chain  | Enables independent processing of each document. Can handle large datasets by reducing them into manageable chunks.                   | Requires careful management of the process. Optional compression step can add complexity and loses document order.                                    |
| Map Re-rank Documents Chain | Provides a confidence score for each answer, allowing for better selection of responses.  | The ranking algorithm can be complex to implement and manage. May not provide the best answer if the scoring mechanism is not reliable or well-tuned. |

You can read more about how to implement different document chains in [Lang-Chain's comprehensive API](#) and [here](#).

Also, it's possible to simply change the chain type within the `load_summarize_chain` function:

```
chain = load_summarize_chain(llm=model, chain_type='refine')
```

There are newer, more customizable approaches to creating summarization chains using LCEL, but for most of your needs `load_summarize_chain` provides sufficient results.

## Summary

In this chapter, you comprehensively reviewed the LangChain framework and its essential components. You learned about the importance of document loaders for gathering data and the role of text splitters in handling large text blocks.

Moreover, you were introduced to the concepts of task decomposition and prompt chaining. By breaking down complex problems into smaller tasks, you saw the power of problem isolation. Furthermore, you now grasp how prompt chaining can combine multiple inputs/outputs for richer idea generation.

In the next chapter, you'll learn about vector databases, including how to integrate these with documents from LangChain, and this ability will serve a pivotal role in enhancing the accuracy of knowledge extraction from your data.