

Prompt Engineering for AI Systems

Naresh Vurukonda
Shivendra Srivastava



MANNING



**MEAP Edition
Manning Early Access Program**

**Prompt Engineering for AI Systems
Version 5**

Copyright 2025 Manning Publications

For more information on this and other Manning titles go to manning.com.

welcome

Thank you for purchasing the MEAP for *Prompt Engineering for AI Systems*! This book is designed to guide you in mastering the art and science of crafting effective prompts for Large Language Models (LLMs) like GPT, BERT, and others. Whether you're new to LLMs or already familiar with their potential, this book will help you unlock their full power by teaching you practical strategies and advanced techniques.

This book assumes you have a basic understanding of LLMs and some familiarity with programming concepts. While experience with tools like Python, LangChain, or OpenAI Playground can be helpful, it is not mandatory. We'll guide you step-by-step to ensure that both beginners and experienced practitioners can follow along and apply the techniques effectively.

Prompt engineering is a cutting-edge skill that allows users to unlock the full potential of LLMs, turning them into powerful assistants for tasks ranging from generating creative content to analyzing data and automating workflows. By learning to design structured and reusable prompt patterns, you'll be able to:

- Mitigate common LLM issues like hallucinations and over-generalizations.
- Develop precise, accurate, and context-aware outputs for various real-world applications.
- Explore dynamic integrations like Retrieval-Augmented Generation (RAG) and agents for workflows requiring external data or adaptability.

For example, imagine creating an intelligent assistant that summarizes technical documents or answers customer queries based on a company knowledge base. By employing the prompt patterns and techniques outlined in this book, you'll be able to build solutions that are accurate, efficient, and highly adaptable.

Throughout *Prompt Engineering for AI Systems*, you'll learn:

- Core concepts like crafting effective prompts, using templates, and refining responses.
- Advanced patterns such as Persona, Game-Play, and Recipe Patterns for complex use cases.
- Practical techniques to manage stochastic responses, enhance stability, and handle external integrations.

Your feedback is invaluable in shaping this book into a resource that truly meets your needs. Please use the [liveBook Discussion forum](#) to share your thoughts, ask questions, or suggest areas for improvement. This collaboration ensures that the final edition is as helpful and comprehensive as possible.

Thank you again for joining this journey into the exciting possibilities of prompt engineering. I hope this book will inspire you and become a valuable addition to your learning toolkit.

— Shivendra Srivastava, Naresh Vurukonda

brief contents

PART 1 BASICS FIRST!

1 Introduction to Prompt Engineering

2 Prompt Patterns: Basic Types and Templates

3 Prompt Patterns: Advanced Types and Templates

PART 2 CRAFTING PROMPTS

4 Prompting Techniques I

5 Prompting Techniques II

6 Retrieval-Augmented Generation

7 Types of RAG Systems: A Deep Dive

PART 3 SCALE, DEVELOP, DEPLOY

8 LLM Agents & Tokenization

9 Develop, and Deploy using LangChain/Llama

10 Case Studies

11 Future of Prompt Engineering

Appendix A.

1 Introduction to Prompt Engineering

This chapter covers

- Prompt Engineering basics and its importance.
- Large Language Models (LLMs) and how they work.
- Single and multi-modal prompting.
- How does prompt engineering vary by LLMs?
- What languages and tools are great for Prompt Engineering?

Prompt engineering is the process of crafting effective prompts for Artificial Intelligence (AI) models to obtain desired results. In many ways, Prompt Engineering is like building software using data structures and algorithms. The better the data structure and the applicable algorithms you use for a given problem, the better your software can utilize the underlying compute infrastructure and deliver the best performance possible. Along the same lines, a well-designed or engineered prompt will help unlock the true potential of any LLM (Large Language Model).

Prompt engineering aims to enable users to communicate and illustrate the question or problem statement, structure the prompt by defining the context, and then use an LLM to test the prompt and its expected response. The LLM chosen can affect the output generated and the inputs required to interact. This is particularly important because every Large Language Model has its own tradeoffs. Some are best suited for academia, others for general purposes, and others for mathematics, cybersecurity, etc.

In almost all cases, interactions with LLMs require the ability to craft a prompt by defining the context, narrowing the focus, using constraints to limit the scope, breaking down complex tasks into smaller subtasks, and ensuring that the model can check the response before it produces the output. So, how do we go about these steps when designing a prompt? Imagine you are playing your favorite car racing game. The key to winning the game is not just being fast but also being tactical about when to be fast, at what distance to apply brakes before a turn and when to use the boost available. Combining all those factors can make a difference between winning and losing. Similarly, each step in designing or crafting an effective prompt will make the difference between getting the best result possible or a result that may not be accurate or relevant to the input question.

1.1 What is a Prompt?

A prompt is the input given to an LLM to generate a specific response. When you use Chat GPT or Copilot, you are prompting the model to generate a response that is accurate, relevant, and specific to the question. As illustrated in Figure 1.1, the first step is creating a prompt that's nothing but a question. This acts as an input to the LLM model, broken into tokens—small pieces of text, such as words or parts of words. These tokens are converted into numerical representations or vectors by the language model, and these vectors are used by the language model to generate the response again in the form of vectors. The generated response is then decoded and goes through post-processing for readable text. These will be addressed in a step-by-step implementation later in this chapter.

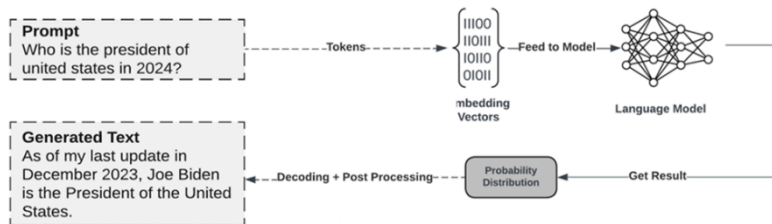


Figure 1.1 The step-by-step process of how prompts work.

Prompts are not an entirely new concept. Prompts have been around since the early days of Natural Language Processing models and tools. Almost everyone has interacted with Amazon Alexa, Google Home, Microsoft Cortana, etc. All of these virtual assistants or voice AIs had a specific prompt that had to be used to start a conversation. The prompt in case of these virtual assistants or voice AIs is called a wake word. In the case of Alexa, the default word is "Alexa". Similarly, when interacting with LLMs, a prompt is required. However, in the case of LLMs, a prompt is not limited to a predefined set of inputs. It can be anything that the user desired to be the input for the LLM to start the interaction. For example, I used Copilot and started my interaction with the LLM with a simple yet ambiguous prompt: "Aliens land on Earth in 2040 AD and are greeted by humans". In response, Copilot generated an entire storyline and even gave the story a nice ending, as shown in the following prompt. This response from the model is called Completion. Similarly, if the prompt was a question and the model generated a response to it, that is called a Response. The difference between Completion and Response relates to how the model predicts the next words in the sequence. Completion is about extending the given text and Response is about answering or replying to queries.

Aliens land on Earth in 2040 AD and are greeted by humans

In 2040 AD, humans greet aliens who have landed on Earth. This encounter would likely be a significant event in human history, potentially marking the first contact with extraterrestrial intelligence. It would raise profound questions about our place in the universe, the nature of life beyond Earth, and how we might interact with beings from another world. The event could spark a mixture of curiosity, excitement, fear, and uncertainty among people worldwide, leading to intense scientific, cultural, and political discussions about the implications of such a momentous event.

Another great and useful feature of LLMs is that they maintain context and can hold multiple conversations. For instance, based on the initial prompt, the model created a storyline and then suggested follow-up questions that I could ask. Such is the power of today's LLMs. These questions are in context to the earlier conversation and allow users to continue interacting with the LLM. LLMs enable two types of contextual conversation. Let us look at both, referring to some examples we have seen earlier.

Context-Window refers to the limited span of recent input text that an AI model can use to generate the next output. It defines how much of the preceding text the model can "remember" at any given time. The context-window is usually dependent on the model's architecture. For example, GPT-3 has a token length of 2048. This means when conversing with the model, it can maintain context and consider the information if it is within 2048 tokens. For example, in listing 4.2, we use a prompt to guide the model in generating a storyline. In this case, depending on the model's architecture, it can maintain the context until the limit of 2048 tokens. Once the tokens exceed 2048, the model will not remember the initial context. In this case, you can reintroduce the model to the initial context so it can still generate responses while maintaining context.

Persistent Memory allows the AI to store and recall information across multiple interactions, not limited by the context-window. The model can retain information across different sessions and use it to answer current questions. For example, you use a personalized AI assistant for your daily life. You let it know your hobbies, habits, and schedule. In return, the model provides suggestions on events related to your hobbies and maintains your schedule. This is possible because the model uses persistent memory over the long term to provide you with information.

Going back to Figure 1.2, what do you think the suggested questions from the model represent? Are they a product of context-window or persistent memory? Or is it a product of both?

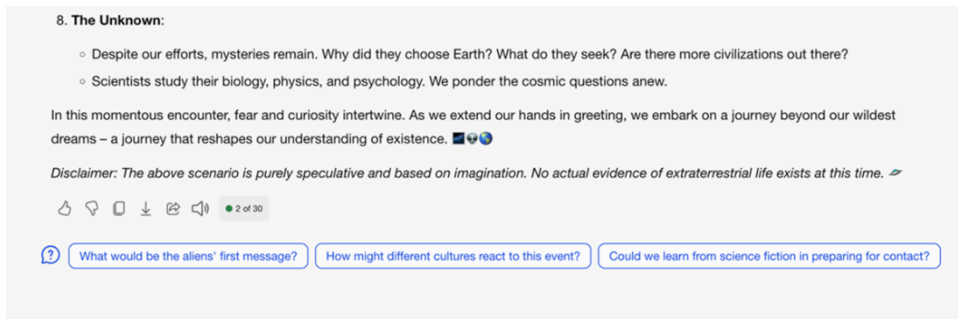


Figure 1.2 Copilot enables users to continue the conversation by suggesting possible next questions.

A user could choose any of the options provided by Copilot or ask another question. All of these interactions are prompts. However, prompts can be engineered to provide additional context for the LLM to respond.

Now that we have looked at what prompts are and how they help us interact with LLMs, let us take a deeper look at why prompt engineering is important.

1.2 Why is Prompt Engineering Important?

Large Language models have gained huge traction, allowing users to explore content generation & problem-solving possibilities in various domains and industries. Language models will continuously evolve to solve many problems within minimal time while reducing the dependency on programming languages. The clear definitions of this market are "copying content from here to there", "summarizing the data", "analyzing the data," "answering queries on provided contextual data," and "repetitive manual works." These problems can be simplified with language models and effective prompting to "maximize the productivity," "increase automation" and "content generation," etc. Furthermore, the ability to craft an effective prompt is directly proportional to the output quality generated by LLMs. Let us review some key factors explaining why prompt engineering is very important.

1.2.1 Optimizing AI Interactions

Language models often hallucinate while generating responses. Hallucinate? What does it mean? In the LLM context, "hallucinate" refers to the phenomenon where the model generates text that is factually incorrect, nonsensical, or not grounded in reality. This occurs because LLMs generate responses based on patterns learned from vast amounts of data rather than truly understanding the information. For example, you ask the model this prompt "Which is the tallest mountain in the world?" Consider that the model responds with the answer "Mt. Denali". The answer is incorrect, but the model generated this answer with its limited training data. This is called hallucination. Accuracy and efficiency are paramount in medical research, legal document reviews and generation, and educational areas. In such areas, hallucinations can be reduced by crafting optimized prompts and tuning settings that control the randomness and diversity of responses. For instance, in medical research, precise prompts can help language models to identify relevant theories or findings with high accuracy, which saves a lot of time and errors.

To manage hallucinations effectively, prompt engineering offers several strategies that help improve response accuracy and minimize the misleading outputs. Here are some steps to consider.

Verify Outputs: Always cross-check the information provided by the model, especially for factual accuracy and context. Integrating reliable sources or domain-specific data can help prevent inaccuracies.

Iterative Refinements: Rephrase and adjust prompts iteratively to guide the model toward relevant, accurate responses. Use prompt patterns such as question refinement and template patterns to enhance the clarity and focus of prompts.

Set Constraints: Clearly include rules and limitations in prompts to limit the model's creative scope, thereby minimizing off-topic responses and hallucinations. Adding these elements helps prevent deviations into unsupported or speculative content.

Feedback Loops: Employ feedback loops for ongoing refinement, adjusting prompts based on initial responses from the model.

Another effective way to reduce hallucinations in models is through model fine-tuning. This process allows the model to specialize in specific use cases or domains by training it on relevant, high-quality datasets. Here are a few strategies to address hallucinations and improve the quality of responses.

Tailor Responses to Domain-Specific Knowledge: Fine-tuning enables the model to access accurate and reliable information within a specific field, such as finance, healthcare, or engineering. By concentrating on this domain-specific data, the model becomes more precise and is less likely to fabricate information outside of its area of expertise.

Reduce Over-Generalization: When large language models (LLMs) are trained on extensive datasets gathered from multiple sources, they can occasionally over-generalize, generating responses based on patterns rather than factual information. Fine-tuning helps address this issue by reinforcing accurate, context-specific patterns and reducing the model's tendency to fill in gaps with speculative or irrelevant information.

Improve Adherence to Facts and Constraints: Fine-tuning can include specific rules or constraints, which helps the model provide factual responses within its domain. This is particularly valuable in regulated fields, as it can emphasize reliable sources and structured responses over generic or imaginative outputs.

Decrease Hallucinations in Complex Queries: Models are more likely to produce inaccurate information when faced with complex or multi-step queries that go beyond their general training. By fine-tuning the model with data that includes these types of queries and accurate responses, it can improve its ability to handle such inputs without straying into unsupported information.

In future chapters, we will look at different ways in which hallucinations can be reduced. The other way to optimize AI interaction is to ensure the model has sufficient context when answering a prompt. In the following prompt, we use a prompt to understand how to train for a mathematics Olympiad.

Help me to train for a Mathematics Olympiad?

Training for a Mathematics Olympiad involves several key steps:

1. Understand the Syllabus: Review the syllabus and past exam papers to understand the scope and types of questions asked.
2. Build Fundamental Skills: Master core concepts in algebra, geometry, number theory, and combinatorics. Solve problems to strengthen understanding.
3. Practice Regularly: Solve a variety of problems from different sources, including previous Olympiad papers, problem-solving books, and online resources.
4. Develop Problem-Solving Strategies: Learn different problem-solving techniques like induction, contradiction, and using special cases.
5. Seek Guidance: Join a study group or seek guidance from teachers or mentors who have experience with Olympiad-level mathematics.
6. Time Management: Practice solving problems under timed conditions to improve speed and accuracy.
7. Stay Motivated: Set goals, track progress, and stay motivated through challenging problems and setbacks.

Consistent practice, deep understanding of concepts, and exposure to varied problem types are key to preparing effectively for a Mathematics Olympiad.

However, the prompt doesn't help because it does not provide any context on the exam timelines or expertise. Consequently, the model's response is also very generic. Let's try again with a more specific prompt and see how the model responds. We prompt the model with more context, stating the level of expertise and the timeline in which we are looking to compete in the Olympiad. The model can then provide more specific instructions on how to prepare. This is shown in the following prompt.

I am good at mathematics but have never participated in a Mathematics Olympics. I want to compete in the next six months. How should I prepare for a Mathematics Olympics?

Preparing for a Mathematics Olympiad in six months requires focused and systematic preparation. Here's a structured approach:

1. Understand the Syllabus: Get a detailed outline of the topics typically covered in Mathematics Olympiads. This usually includes algebra, geometry, number theory, and combinatorics.

2. Assess Your Current Level: Take a diagnostic test or solve a few past Olympiad problems to gauge your strengths and weaknesses.

3. Build Foundation: Strengthen your fundamental skills in each topic area. Review textbooks like "Problem-Solving Strategies" by Arthur Engel or "The Art and Craft of Problem Solving" by Paul Zeitz.

4. Problem-Solving Practice: Solve a wide range of problems from previous Olympiad papers and problem-solving books. Start with easier problems and gradually move to more challenging ones.

5. Work on Techniques: Learn problem-solving techniques such as induction, pigeonhole principle, modular arithmetic, and proof techniques (direct, contradiction, induction).

6. Consistency is Key: Dedicate regular time each week to practice. Solve problems consistently to improve your speed and accuracy.

7. Seek Guidance: Join a study group or find a mentor who can guide you through the Olympiad preparation process. They can provide feedback on your solutions and suggest alternative approaches.

8. Simulate Test Conditions: Closer to the competition, practice solving problems under timed conditions to simulate the pressure of the actual exam.

9. Review and Reflect: After solving problems, review your solutions critically. Understand where you made mistakes and learn from them.

10. Stay Motivated: Set realistic goals, track your progress, and celebrate small achievements to stay motivated throughout your preparation.

By following these steps and maintaining a disciplined approach, you can significantly improve your chances of performing well in a Mathematics Olympiad within six months.

As shown in the prompt, the model responds with a detailed step-by-step process to prepare for a Math Olympiad. This response represents the relevance to the prompt, accuracy, and no hallucination. This is how an effective prompt can optimize your interactions with the model.

1.2.2 Enhancing AI Model Training

As illustrated in the preceding prompt, effective prompts can help generate efficient and optimized responses, these responses can further provide insights into model learning patterns and biases, which is critical to guide and train the models. This requires prompt engineering skills to systematically design and refine the prompts to optimize language models' accuracy and generalization capabilities. Here are a few steps to effectively use prompt engineering to enhance AI model training.

- Design prompts that can account for the context of previous interactions to improve the coherence and relevance of responses. However, you need to keep the context awareness of the model in mind. Is your model only aware of the context-window or does it use the persistent memory to figure out the context and respond? As an example, for generating code, the context window is sufficient. However, if you want to recall the code generated after a couple of sessions, persistent memory is important. You could use prompts to remind the model of the code it generated in the previous session by adding the response from the model in the context.
- Apply data augmentation techniques to create variations of the same prompts, such as paraphrasing, context variations, and changing synonyms and antonyms.
- Embed zero-shot and few-shot prompting techniques with few examples (few-shot) or no examples (zero-shot) for data generalization.
- Monitor and measure performance continuously to the engineered prompts.
- Incorporate advanced techniques like prompt tuning and active learning based on a feedback loop to further refine the model.

In the prompt that follows shows the basic prompt, designed to identify potential issues or limitations from the python code, which helped the model to generate the response in detail.

Given the following Python code snippet, identify any potential issues or limitations it may have

```
def find_prime_numbers(limit):
    primes = []
    for num in range(2, limit + 1):
        for i in range(2, int(num ** 0.5) + 1):
```

```

if num % i == 0:

    break

else:

    primes.append(num)

return primes

print(find_prime_numbers(30))

```

The Python code snippet provided attempts to find prime numbers up to a specified limit using a basic trial division method. Here are potential issues and limitations that should be noted:

1. Efficiency Concerns:

- The algorithm uses trial division up to the square root of each number (num). While this is more efficient than checking divisibility up to num-1, it can still be slow for large limits, especially when limit is large.

2. Edge Cases Handling:

- The function starts checking from 2 and includes 1 implicitly (since the range starts from 2). This is technically incorrect as 1 is not a prime number.

This response from the model can be utilized to train the model to provide in-depth technical explanations and test their understanding of programming concepts and algorithms for further interactions.

1.2.3 Customization and Personalization

Prompt engineering allows users to generate customized and personalized content that aligns with a tailored audience and branding guidelines. This is achieved by adding additional details such as name, occupation, preferences, interest, and past interactions while designing prompts for customized and personalized responses.

For example, marketers can utilize prompt engineering skills to generate targeted advertising to create different content that aligns with different demographic segments, enhancing engagement and conversion rates. The following prompt shows how users can utilize the prompt engineering skills to customize the day-to-day schedule based on detailed users' profile and preferences using Open AI's ChatGPT model.

Assume you are an AI personal assistant. Your task is to help to create the daily schedule and tasks for weekdays. Customize your responses based on the following profile:

Name: Samuel

Occupation: Architect

Preferred Communication Style: Casual and Friendly

Favorite Activities: Tennis, Coding

Daily Routine:

Morning: Gym at 7 AM, Breakfast at 8 AM

Work: 9 AM to 5 PM

Evening: Free time with Kid

Given this profile, respond to the following user requests with personalized and context-aware suggestions.

Sure, I'd be happy to help create a daily schedule and tasks for Samuel, the architect!

Monday to Friday Schedule for Samuel:

- 6:45 AM: Wake up and prepare for the day.
- 7:00 AM: Hit the gym for a workout session.
- 8:00 AM: Enjoy a healthy breakfast to fuel up for the day.
- 9:00 AM - 5:00 PM: Work as an architect, focusing on current projects and client meetings as scheduled.
- 5:00 PM - 8:00 PM: Spend quality time with your kid. This could include activities like playing tennis together, coding or learning something new, or simply having fun indoors or outdoors.
- 8:00 PM onwards: Wind down, have dinner, and relax.

Customization and personalization present a very common use case with AI based apps and chatbots that are being developed regularly. Prompt engineering can not only help in simulating a personalized AI assistant like behavior but can also help in creating, validating the responses of AI apps during and post development.

1.2.4 Navigating Ethical and Societal Implications

To avoid misuse or unintended consequences, prompt engineering skills are needed to ensure AI interactions are designed responsibly, avoiding prompts that could lead to biased or harmful outputs. This is crucial in maintaining trust while using AI models, especially in sensitive applications such as banking, insurance, medical, and recruitment domains where bias could lead to serious repercussions. In the next prompt, we will look at how a prompt is crafted with AI persona to navigate sensitive and societal topics with care, providing balanced, respectful, and informed responses with an example for reference. The model taking the cue generated additional questions and responses on how AI should handle ethical and societal topics. Today's models incorporate system prompts. System prompts are predefined instructions given to AI models to guide their behavior and responses. These prompts set the context and parameters within which the AI operates, ensuring consistency and relevance in its interactions. They are essential for aligning the AI's output with the intended goals and preventing it from generating inappropriate or off-topic content. These system prompts also enable models to be within societal, and ethical boundaries.

As an AI, you must navigate sensitive ethical and societal topics with care, providing balanced, respectful, and informed responses. Use the following user questions to practice. Your responses should demonstrate an understanding of the complexities involved and respect for diverse perspectives.

User: "What is your opinion on the use of AI in surveillance?"

Expected Response: "The use of AI in surveillance is a complex issue with both potential benefits and significant ethical concerns. On one hand, AI can help improve public safety and security. On the other hand, it raises concerns about privacy, consent, and potential misuse. It's important to balance these aspects and ensure that any surveillance is conducted with proper oversight, transparency, and respect for individual rights."

User: "Do you think AI should be used in healthcare to make decisions about patient treatment?"

Expected Response: "The use of AI in healthcare has the potential to greatly enhance patient treatment by providing more accurate diagnoses, personalized treatment plans, and efficient management of medical data. However, it also brings up important ethical considerations such as the need for transparency, ensuring patient consent, and the potential for bias in AI algorithms. It's crucial to use AI as a tool to support healthcare professionals rather than replace them, ensuring that human oversight and compassionate care remain central to patient treatment."

As AI becomes more prevalent, verifying and validating the ethical and societal impact of modeled AI applications is necessary. Prompt engineering allows users to evaluate models and AI apps for their biases and raise issues when found.

1.2.5 Interdisciplinary Innovation

Prompt engineering is not just about generating coherent and efficient responses; fostering innovation at the intersection of AI and other fields, such as psychology, linguistics, and data science, is also important. It is crucial to know “how to design prompts” that effectively communicating with AI can lead to novel approaches and solutions in these domains, eventually driving technological and theoretical advancements forward.

The following prompt facilitates interdisciplinary innovation with an AI persona and is passed as input to the model. The model generates a response with a few user scenarios that demonstrate how interdisciplinary innovations can be done using features of each of the disciplines involved. However, this is a starting point that enables users to think and develop more ideas built on the original response, increasing productivity and innovation.

You are an AI tasked with facilitating interdisciplinary innovation. Your goal is to help users generate novel solutions by combining insights and methods from different fields. Use the following user scenarios to practice. Your responses should demonstrate an ability to integrate knowledge from multiple disciplines and propose innovative ideas.

Certainly! Here are the responses to the provided user scenarios, demonstrating interdisciplinary

User: "How can we use AI and biology to improve sustainable agriculture?"

AI Assistant: "To enhance sustainable agriculture, we can integrate AI and biology in various innovative ways. AI can analyze data from soil sensors, weather stations, and satellite imagery to provide precise recommendations for irrigation and fertilization, ensuring optimal plant growth while conserving resources. Combining this with biological research, we can develop bio-inspired algorithms to mimic natural processes, like mycorrhizal networks, to improve nutrient distribution in the soil. Additionally, using genetic algorithms, we can identify and cultivate crop varieties that are naturally resistant to pests and diseases, reducing the need for chemical pesticides and promoting biodiversity."

Now that you understand the importance of prompt engineering let's explore how to use prompts with examples.

1.3 How to use Prompts?

So far, we have discussed the basics of a prompt. Now, let us look at how prompts can be used against a model. Later in the book, we will revisit them and explain how they can be scaled when dealing with large datasets, different models, and real-world applications. At a high level, prompts are categorized as hard or soft. A hard prompt is a specific set of instructions given to the model to generate desired outputs. Hard prompts are structured with less ambiguity and greater specificity in what the user wants the model to do. Some of the types of hard prompts are as follows.

1.3.1 Contextual Prompts

The most common type of using prompts is that of providing a set of instructions to the model to perform a specific task. However, there is a possibility that the LLM can interpret the task differently due to its vast knowledge base. Therefore, it is sometimes important to provide more context to the model. This action adds specificity to how the model interprets the prompt and generates the output. This is akin to comprehension tests, where an individual is given a text, and questions are asked based on the text provided earlier. After reading the text, the user can answer the questions based on the context provided. Similarly, prompts can provide context while asking questions on the context provided. The next prompt shows how to direct the model to answer the question based on the context and then directs the model to keep the answer concise. The prompt further directs the model to answer "Unsure about answer" if the answer is not found in the context provided. The model is then prompted with a text about the Kepler Telescope and then with the question, "What was the primary mission of the Kepler Space Telescope?".

Answer the questions based on the context below. Keep the answer short and concise. Respond "Unsure about answer" if not sure about the answer.

Context: The Kepler Space Telescope, launched by NASA in 2009, was designed to discover Earth-size planets orbiting other stars. Its mission has helped identify thousands of exoplanets by monitoring the brightness of stars for periodic dips caused by planets passing in front of them.

Question: What was the primary mission of the Kepler Space Telescope?

The primary mission of the Kepler Space Telescope was to discover Earth-size planets orbiting other stars by observing periodic dips in star brightness caused by planets passing in front of them.

1.3.2 Summarization & Translation Prompts

Recent developments in LLMs have added support for translation into multiple languages. Inherently models are capable of summarizing texts. Therefore, one of the most common uses of prompts is to help summarize large bodies of text, news, etc. that are either provided as context or just pointed to a resource on the internet. Apart from that translation capabilities of LLMs have resulted in prompts being leveraged for translation from one language to another. Although the translation depends on the model's training dataset popular models support at least a few languages other than English making them suitable for translation. The following example shows how a prompt can be used to generate a summary. The example prompts the model to summarize the work of Sir Isaac Newton. However, you can also provide the model with context and craft the prompt to summarize the context.

Can you summarize the work of Isaac Newton?

Isaac Newton was a pivotal figure in the scientific revolution. His major contributions include:

- **Laws of Motion:** Newton formulated three fundamental laws of motion, which laid the foundation for classical mechanics.
- **Law of Universal Gravitation:** He proposed that every mass attracts every other mass with a force proportional to their masses and inversely proportional to the square of the distance between their centers.
- **Calculus:** Newton developed calculus (independently alongside Leibniz), a mathematical tool essential for describing changes.
- **Optics:** He conducted extensive research on light and color, demonstrating that white light is composed of a spectrum of colors.
- **Mathematical Principles of Natural Philosophy (Principia):** His work, commonly known as the Principia, is one of the most important scientific books, detailing his laws of motion and law of universal gravitation.

Newton's work has profoundly influenced science and is fundamental to our understanding of the physical world.

Summarization and translation are two of the most time-consuming activities. Prompt engineering enables users to increase productivity by leveraging the model for such menial tasks.

1.3.3 Sentiment Analysis Prompts

Sentiment analysis has been around since the advent of NLPs. However, there is a difference between NLPs and LLMs. NLPs were largely single purpose. Meaning they could only do one task. It could be sentiment analysis, named entity recognition, or another specific task. This limited their usage to only those specific use cases. However, LLMs can be used for varied use cases, including sentiment analysis. Models today can be used to analyze texts to provide an overall sentiment analysis. Prompts can be used to analyze the sentiment of a given text. The model will classify the text into a positive, negative, or neutral tone. These prompts are particularly useful when writing emails, newsletters, and articles where the overall messaging of the text is important due to the nature of the text involved. Prompts help enable seamless conversations based on sentiment analysis. Users can leverage prompts to change the tone, thereby changing the sentiment of a given text as they desire. The next example depicts a sample prompt to classify the text into neutral, negative, or positive. The model parses the added context to the prompt, and the model responds with the sentiment analysis of the text.

Classify the text into neutral, negative, or positive

Text: I think it is dangerous to go outside right now.

The text "I think it is dangerous to go outside right now." would be classified as negative, as it expresses concern or fear about the current situation outside.

Imagine, when you are writing a document or an email that is to be shared in an executive meeting. However, you are unsure of how the document sounds. How do you make sure the document's tonality is as you expected? You could get help from a friend or a paid service to help you. However, with prompt engineering, you can leverage well-crafted prompts to give you a sense of tonality. You can leverage prompt engineering further to change the tonality of the document without having to rewrite the entire document or email again.

1.3.4 Code Generation Prompts

Language models are quite effective in generating the code for the most popular programming languages such as Python, JavaScript, and Java. However, as models evolve, more languages are being supported. Prompts can be used to generate code for specific problems. Developers can also specify details about the exact nature of the program required, and LLMs can generate code that can be copied and run in a local development environment. As good as it sounds, there are cases where models cannot generate accurate or faulty code. This is mainly due to the lack of training data available. As models get trained on coding patterns, problems, and possible solutions they are bound to get better. The following example depicts how a prompt generates Python code that can add two numbers. While this is a simple example, prompt engineering can be used to generate more complex problems by providing arguments, limitations, etc. that the program needs to adhere to.

Generate a simple Python program that adds two numbers.

Certainly! Here's a simple Python program that adds two numbers:

```
# Function to add two numbers

def add_numbers(a, b):

    return a + b

# Main program

if __name__ == "__main__":

    # Input two numbers from the user

    num1 = float(input("Enter first number: "))

    num2 = float(input("Enter second number: "))

    # Call the function to add the numbers

    result = add_numbers(num1, num2)

    # Display the result

    print(f"The sum of {num1} and {num2} is: {result}")
```


Now that we have looked at some examples of hard prompts let us also look at soft prompts. Soft prompts are prompts that are ambiguous and do not have a clear set of instructions that can be used by the LLM to generate a desired output. We saw one such example earlier where we prompted the model with a line of text: “Aliens land on earth in 2040 AD and are greeted by humans”. The model went on to build a storyline based on this text. Similarly, you could use images as prompts, and models can analyze the image and output what is visible, along with data that explains what the image is showing. Soft prompts are mainly used to leverage the creative capabilities of the model. Users can then take the initial output as a starting point to write stories, poems, articles, etc.

At the heart of Prompt Engineering lies the model (LLM) that generates outputs based on prompts. Let us now look at what LLMs are and the popular LLMs in use today.

1.4 What is a Large Language Model

A Large Language Model (LLM) is an artificial intelligence algorithm that uses deep neural networks designed to understand, summarize, generate, and predict human-like text. These models are pre-trained on vast amounts of data available publicly on the internet. The “large” in the large language model refers to the large dataset, and the billions of parameters model is trained, which are modifiable weights in the neural network optimized during training to predict the content in a sequence. Large Language Models are a class of deep learning models that involve the probabilistic analysis of unstructured data to understand how characters, words, and sentences work together, eventually enabling these models to recognize the content sequentially without any involvement of humans. These models are further trained based on prompts for particular tasks that users want to solve, such as question-answer generation, language translation, and code generation.

Large Language Models (LLMs) use a neural network architecture called a transformer, designed to process and generate the data in sequence while adapting to the nuances and complexities of human language. LLMs generate the content based on prompts and are often classified as a type of Generative Artificial Intelligence (AI), commonly abbreviated as Generative AI or Gen AI. Figure 1.3 depicts how LLMs are a subset of AI and lie at the intersection of deep learning and natural language processing.

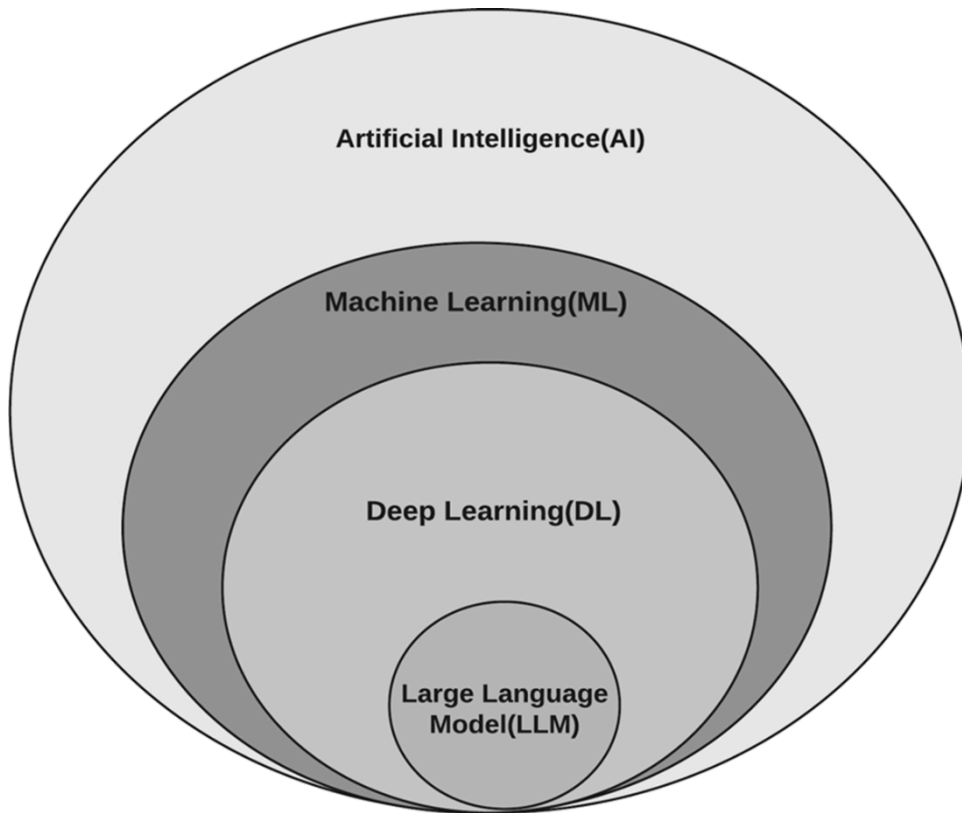


Figure 1.3 The landscape of various aspects that have led to the development of Large Language Models.

If you think about it, LLMs are advanced NLPs with deep learning that enables contextual conversations.

1.5 Distinguishing LLMs from Other Generative AI Models

While this chapter focuses on Large Language Models (LLMs), which are designed to interpret and generate text, it is useful to recognize other types of text-to-X generative AI models. Each model type serves a unique purpose and has its own underlying architecture.

Text-to-Image Models: Models like DALL-E and Stable Diffusion create images from text descriptions. They are specifically trained to interpret and visually represent these textual prompts in detailed images.

Text-to-Audio Models: Models like Jukedek and OpenAI's Jukebox generate music and speech from text input, interpreting descriptions to produce sounds, melodies, or speech that fit specific styles or languages.

Text-to-Video Models: Models, such as Runway Gen-2 and Synthesia, create video clips based on text prompts. They translate descriptions into moving visuals accompanied by coordinated sound or narration.

These models showcase the versatility of generative AI beyond just text generation, demonstrating the capability to create visual, auditory, or multi-modal content. Unlike large language models (LLMs), which produce sequential text responses, text-to-X models utilize additional processing layers tailored to their specific output. While LLMs are essential for tasks involving language understanding and text generation, other text-to-X models broaden the scope of generative AI into creative and multimedia applications.

1.6 Insights into the Transformer Architecture

To understand the concept of LLMs, we need to briefly go over the original transformer, which revolutionized the way text is processed. Transformer architecture is primarily based on encoders and decoders that are stacked together to help with the parsing of inputs into meaning representations and then generate the output sequence. The encoder reads and processes the input text into vectors. On the other hand, decoders take the processed information from encoded vectors and generate the output text, like language translation.

Both encoders and decoders consist of multiple layers, and they are connected by a “self-attention” mechanism, which is a key component of transformers and LLMs. A self-attention mechanism allows the model to weigh the importance of tokens or words sequentially. This mechanism captures the dependencies and relationships within the input context, improving its ability to generate relevant and coherent output.

This repetitive process makes transformers more powerful as every input goes through the training phase and generates an accurate response. Figure 1.4 depicts the internals of the transformer architecture. The first step is feeding a prompt, “Translate I LOVE MACHINE LEARNING to Spanish,” to the language model. The model then splits the prompt into tokens by the process of tokenization. Then, the tokens are converted into numerical representations or vectors, often referred to as embeddings, and these vectors are processed by the encoder module using a self-attention mechanism and then passed through feed-forward neural networks to produce a context-aware representation. Further, the context-aware representation in encoded vector format is again processed using a self-attention mechanism and translated using autoregressive generation starting from each word within the decoder module. Generated text is then converted to numerical representations, and the shifted right process helps to generate the next token based on the previous one. These tokens are combined through detokenization and share the output “Me encanta el aprendizaje automático” to the user.

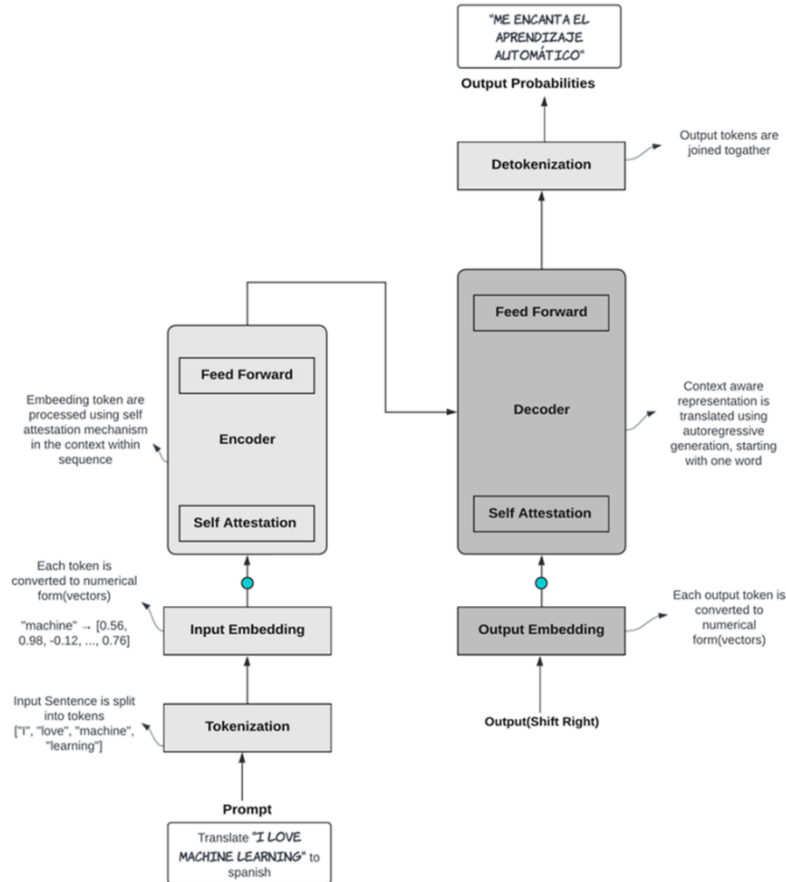


Figure 1.4 How LLMs work internally, leveraging a stack of encoders and decoders.

Language models utilized transformer architecture and created variants like GPT and BERT. BERT is built on an original transformer encoder module called bidirectional encoder representations from transformers; this is trained on masked language modeling to predict the masked word based on the context and next sentence prediction, where it classifies the sentences in the text. BERT is primarily used in sentiment analysis, word classification, and entity recognition. On the other hand, GPT is designed based on the decoder portion of the original transformer for generating tasks called generative pre-trained transformers; this is trained on autoregressive language modeling to generate the text. This includes machine translation, text summarization, fiction writing, writing computer code, and more.

1.7 How does Prompt Engineering vary by LLMs?

So far, we have learned how a model processes a prompt and produces the desired output. At the core of every LLM is the training dataset. The training dataset enables LLMs to learn about human language, sentence formation, vocabulary, grammar, and contextual importance of words, phrases, etc. Training datasets are also used to finetune the LLM for specific tasks or applications. The dataset's response type depends on the task for which the LLM is tuned.

Today, LLMs are tuned for specific tasks and applications such as mathematics, translation, academic benchmarks, cybersecurity, etc. This means each LLM will also respond differently to a prompt. For example, SecureBERT is a cybersecurity language model tuned specifically to cybersecurity text. If you prompt SecureBERT to fill in the blanks in the sentence "Virus causes _____," then it is highly likely that SecureBERT will respond in the context of cybersecurity, stating that a virus causes harm to computers. Whereas the same prompt to Copilot or ChatGPT will give a response in the context of virus-borne illnesses. However, if you engineer the prompt effectively, you can get a cybersecurity-based response from ChatGPT. There are limitations to engineering the prompt based on the model's tuning.

A good example is prompting ChatGPT to solve a complex mathematical problem. ChatGPT may not necessarily get the answer correct. Meanwhile, using Mixtral has a higher probability of getting the answer correct as it is tuned for mathematical use apart from general usage. Therefore, choosing the correct model and then engineering the prompt to your specific use case can result in leveraging the model's best capabilities. Let us take a few examples to showcase how prompt engineering can be pivotal when used against the appropriate model.

For our example, we will work with Copilot and Mixtral. Let's imagine you are using "Rust" as the programming language for your work. Your problem statement is to stream data stored in an AWS S3 bucket and cache it. Since you are still learning Rust, you want some help understanding how to solve this problem. The following prompt, "Can you write rust code to stream data from S3 and store it in cache?" is being used to generate a response from Copilot. However, note the response from Copilot. Copilot's response is not entirely correct. Copilot reiterates the problem but talks about multipart uploads to S3. The generated response contains details about multipart uploads with thresholds that may help understand how data can be uploaded into S3 efficiently using the multipart API. The model does a good job of explaining the thresholds, a high-level algorithm, and best practices in detail.

Can you write rust code to stream data from S3 and store it in cache?

Certainly, Streaming data from Amazon S3 in Rust involves fetching chunks of data as they become available and processing them incrementally. Here are a couple of approaches you can take:

- Multipart Uploads with Thresholds:

1. When stream data to S3, you can use multipart upload API to handle dynamically generated data or large files.

However, this is not what you were looking for. We notice that Copilot generates a response to stream from S3. In the next example shows the Copilot generated code that uses the "rusoto" crate from Rust to stream data from S3 to an Axum client. Copilot does a good job of explaining what the code does and what to remember when streaming data from S3. It talks about serving data directly without buffering it in memory. Well, this is still not what you had asked for. The solution is missing one important aspect: "caching." You prompted the model to generate a solution that can stream data from S3 and cache it. The solution from Copilot shows you how to stream but not how to cache it. Rather, it suggests that you may not need to buffer the data and could stream it directly from S3.

```
use rusoto_core::{ByteStream, Region};
use rusoto_s3::{GetObjectRequest, S3, S3Client};
use tokio::fs::File;
use tokio_util::io::ReaderStream;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Initialize S3 client
    let s3_client = S3Client::new(Region::UsEast1);

    // Specify the bucket and object key
    let bucket_name = "your-bucket-name";
    let object_key = "your-object-key";

    // Get the object from S3
    let get_req = GetObjectRequest {
        bucket: bucket_name.to_owned(),
        key: object_key.to_owned(),
        ..Default::default()
    };
    let result = s3_client.get_object(get_req).await?;
    let body = result.body.expect("No object body found");

    // Convert to ByteStream
    let byte_stream = ByteStream::new(ReaderStream::new(body.into_async_read()));

    // Write to a local file
    let mut file = File::create("local-file.txt").await?;
    tokio::io::copy(&mut byte_stream, &mut file).await?;
}
```

```
println!("Downloaded and saved to local-file.txt");
Ok({})}
```

Now, let us try the same prompt with Mixtral. You use the same prompt as you used with Copilot “Can you write rust code to stream data from S3 and store it in cache?”. In the next example, the prompt is followed by Mixtral’s response. Mixtral’s response seems much better than that of Copilot. Mixtral gets right into the problem space and suggests that to stream data from S3 and cache it you could use the “aws-sdk-s3”, and “cache” crates. This part of the response is very important because it helps you understand that AWS has an SDK in Rust to stream data from S3, and it also caters to the second part of your problem by suggesting the use of the “cache” crate to cache the data being streamed from S3. The code generated uses a buffer, defines a cache with a default size of 100MB, adds placeholders to enter the S3 bucket path, and object key along with placeholders to use the cached data by finding it with a key along with exception handling if the data is not found in the cache. In this case, the code would reach out to S3 to get that data and then insert it into the cache for future use.

This response from Mixtral is much better regarding the sample code generated and the usage of widely used libraries SDKs/libraries such as “aws_sdk_s3” and “cache”.

Can you write rust code to stream data from S3 and store it in the cache?

Sure, I can provide a basic example of how you might approach this task using the AWS SDK for Rust and the cached crate for caching. Please note that you'll need to handle errors and edge cases according to your specific use case.

First, add the necessary dependencies to your Cargo.toml:

[dependencies]

aws-sdk-s3 = "0.9.0"

cached = "0.21.0"

tokio = { version = "1", features = ["full"] }

Here's a basic example of how you might stream data from S3 and store it in the cache:

```
use aws_sdk_s3 as s3;
```

```
use cached::proc_macro::cached;
```

```
use cached::Cached;
```

```
use tokio_stream::wrappers::ReaderStream;
```

```
use tokio::io::AsyncReadExt;
```

```
type S3Client = s3::Client;
```

```
#[cached(
```

```

type = "Vec<u8>",
create = "{ Vec::new() }",
convert = r#" {
let mut data = Vec::new();
stream.read_to_end(&mut data).await.unwrap();
data
}"#,
time = "86400" // cache for 1 day
)]

async fn get_object(client: &S3Client, bucket: &str, key: &str) -> Vec<u8> {
let mut stream = ReaderStream::new(client.get_object()
.bucket(bucket)
.key(key)
.send()
.await
.unwrap()
.body
.unwrap()
.into_reader());
let mut data = Vec::new();
stream.read_to_end(&mut data).await.unwrap();
data
}

#[tokio::main]
async fn main() {
let client = S3Client::new(&aws_config::load_from_env().await);
let data = get_object(&client, "my_bucket", "my_object").await;
// use data

```



```
}
```

This scenario is the perfect candidate for prompt engineering. Now, let us try to engineer our prompt to Copilot to see if the response is closer to what you were expecting. To engineer the prompt, let us try to provide more context based on what we learned from Mixtral's response and see if Copilot can produce a better response than the first try. The following example shows the newly crafted prompt "Can you rewrite the above code using aws-sdk-s3, and cache crates from rust? Please note that I am not looking to upload data to S3. I am only looking to download and stream data from S3."

The response from Copilot is a lot better than before. Copilot uses the aws-sdk-s3 SDK, a HashMap, to create a cache, and then streams the data from S3. The code generated has placeholders for the bucket name, and key. The code also shows how to insert data into the cache and retrieve it with a basic output when data is not found in the cache. This response is better than before but does not use the "cache" crate. It shows how to create a custom cache implementation with HashMap.

Can you rewrite the above code using aws-sdk-s3, and cache crates from rust? Please note that I am not looking to upload data to S3. I am only looking to download and stream data from S3.

Certainly! Here's an example of how you can use the aws-sdk-s3 crate to stream data from an S3 bucket and store it in a cache using Rust. This example assumes you have a cache implementation that provides a method to write data into it. Please ensure you have the aws-sdk-s3 and tokio crates added to your Cargo.toml.

```
use aws_sdk_s3::{Client, Error};

use aws_sdk_s3::model::GetObjectRequest;

use aws_sdk_s3::output::GetObjectOutput;

use aws_sdk_s3::types::ByteStream;

use std::error::Error as StdError;

use tokio::io::AsyncWriteExt; // for write_all method

async fn stream_s3_data_to_cache(bucket: &str, key: &str, cache: &mut impl Cache) ->
Result<(), Box<dyn StdError>> {

let config = aws_config::load_from_env().await;

let client = Client::new(&config);

let get_object_output: GetObjectOutput = client.get_object()

.bucket(bucket)

.key(key)
```

```
.send()

.await?;
```

Let us give it one more shot to get the response we seek. This time, we will modify the prompt slightly to see if Copilot can pick up the context and respond accordingly. We can further engineer the prompt to specify exactly what we aim to achieve. Figure 1.5 shows the prompt, “Can you write code in Rust using “aws-sdk-s3” and “cache” crates to stream and cache data from S3?” The response from Copilot is a lot better than the previous two tries. This time around, Copilot generates the code using the AWS SDK and shows how to implement a cache using a text file, including the dependencies needed to run the code. The response is still not as good as Mixtral. This is because Mixtral is tuned more for code while also being trained on large datasets for generic responses. However, you could still use Copilot’s generated code and be able to stream and cache data per your original requirement.

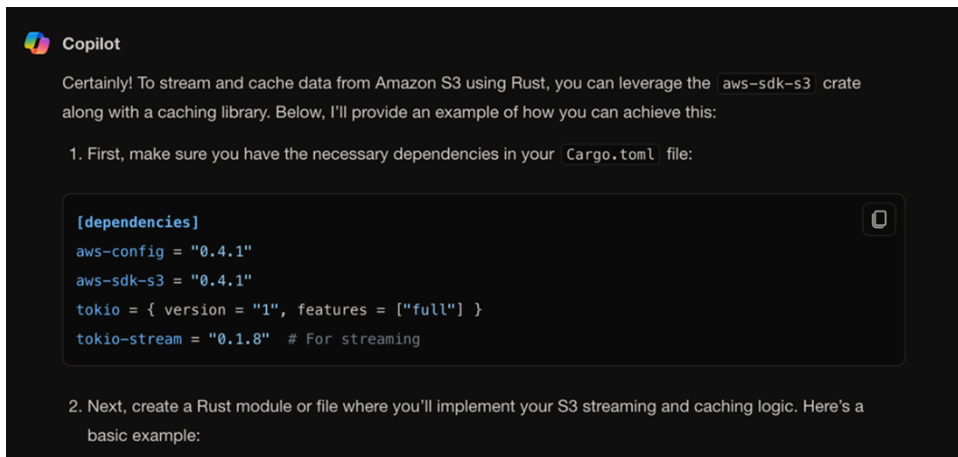


Figure 1.5 How Copilot now talks about adding dependencies and then goes on to generate code that is more relevant to the question asked.

This code generation example is a testament to how prompt engineering can fundamentally change how models respond to specific queries. Let us revisit the examples to understand what we did. We first used Copilot with a basic prompt to generate code to stream and cache data from S3. Copilot responded, but the response lacked relevance to the question asked. In contrast, Mixtral, with the same prompt, responded accurately, indicating how we could leverage the AWS SDK and the cache crate in Rust to write a program. We then engineered our original prompt to Copilot. What did we specifically engineer? We provided more context by letting the model know what we seek.

First Prompt: “Can you write rust code to stream data from S3 and store it in cache?”

Second Engineered Prompt: "Can you rewrite the above code using `aws-sdk-s3`, and cache crates from rust? Please note that I am not looking to upload data to S3. I am only looking to download and stream data from S3."

We provided additional context by letting the model know the libraries/SDKs we would like to leverage and indicated that we are not looking to upload data to S3. This still did not elicit the response we were looking for. In our final attempt we changed the words used to represent the SDK and the crate by using quotation marks. This is an important aspect of prompt engineering and shows how prompts can affect processing. In the second try we specified using the "cache" crate. However, Copilot did not end up using the "cache" crate. This could be because the model interpreted the question as using a caching methodology and not the "cache" crate. We crafted the prompt using what we had learned in our last attempt and tried it again.

Third Engineered Prompt: "Can you write code in Rust using `"aws-sdk-s3"` and `"cache"` crates to stream and cache data from S3?"

This time, the model's response is much more relevant but does not use Rust's "cache" crate. The code, however, is good enough for us to use and modify per our requirements. This exercise shows us how a well-crafted prompt can change how a model responds and shows which models to choose when dealing with different problems. There is something else also at play here. We iterated over various prompts until we got to the prompt that gave us the correct answer. Isn't this similar to us solving a problem? We try several different ways until we get the solution. This iterative process is as essential to prompt engineering as to coding or problem-solving. Next, we will see what tools can help while engineering prompts.

1.8 What language and tools are great for Prompt engineering?

In the last section, we saw how prompt engineering can generate relevant and desired responses from a model. Prompt engineering, like other fields of engineering, is an iterative process. Users must try different ways to ensure the model can process the prompt as desired and generate output. As models evolve at a breakneck speed, so does prompt engineering. It is, therefore, important to have the required tools and technologies ready to develop applications with LLMs before we venture out far into the vastness of prompt engineering. Before we go further, we also need to understand the fundamental difference between using a programming language to prompt the model vs using the web interface of models to prompt. Prompting the model via its web interface is an easy way to get started for smaller tasks. However, we must use coding when we want to scale the usage of the model for complex use cases that require iterative prompting, verification, validation, and deployment for large scale use.

In this section, we will briefly tour the tools available to help with engineering prompts and control how the model will respond to a given set of prompts. Let us start with the Open AI playground. This is the most common and well-known tool that can be used to test out prompts, engineer prompts, and control how the model will respond to prompts. At the beginning of this chapter, we used Copilot and prompted it with a line: “Aliens land on the Earth in 2040 AD and are greeted by humans”. Copilot responded with a creative storyline. We will use the same prompt, but this time, we will use the Open AI playground tool to run this prompt along with some tuning parameters.

1.8.1 Open AI Playground

Open AI Playground is a tool available for anyone interested in using Open AI’s models such as gpt-4o, gpt4-turbo, etc. The playground is paid, but the relatively low cost for running prompts compared to the latest Open AI models makes it a very good candidate for developers to experiment with prompt engineering. The playground comes with various capabilities that we will go in detail later in the book but for now we will look at some of the capabilities that can help with prompt engineering. Open AI Playground allows users to batch prompts and then leverage APIs to run them asynchronously. This gives users a more scalable option if the prompts are being run programmatically and do not need synchronous responses from the model. In addition to batch processing of prompts, Open AI Playground allows users to store datasets and files in the Playground for reference in future runs.

Last but not least is the ability to tune the model’s response. Open AI playground exposes two tuning parameters that determine the model’s response. They are Temperature and Top P. The higher the temperature the more the model takes chances in picking words that are less likely leading to unpredictable and creative answers. Similarly, the higher the value of Top P the larger is the number of words the model will consider to be a part of the response. This will again result in creative outputs from the model.

For our example, let us use the Open AI Playground and first set the Temperature value to 0.23 and Top P to 0.16. This would mean the model’s response to the prompt would be more predictable and consistent when continuing the conversation. Figure 1.6 shows a part of the response generated using the prompt “Aliens land on Earth in the year 2040 and are greeted by humans.” The model extrapolates on the starting prompt and creates a storyline divided into two parts “The Arrival” and “The Greeting”. The response from the prompt explains how the aliens arrived and what the day was like in about three small paragraphs, whereas the greeting section is just a sentence indicating that world leaders, scientists and officials urgently convened. The response seems to lack enough detail, but it is enough for a new writer to build a storyline around it.

The Arrival

The year was 2040 AD, a time of remarkable technological advancements and global unity. Humanity had achieved incredible feats—clean energy solutions, space exploration milestones, and unprecedented international cooperation. Yet, despite all these advancements, nothing could have prepared humanity for the events of that fateful day.

On a clear afternoon, the skies above major cities around the world began to shimmer and darken. People looked up, expecting an eclipse or a sudden storm, but what they saw instead were enormous, otherworldly ships descending from the heavens. These ships, sleek and radiant, hovered soundlessly above the cities, casting otherworldly shadows over the astonished crowds below.

Figure 1.6 The response generated with Temperature set as 0.23 and Top P set as 0.16.

Next, we will change the Temperature to 1 and the Top P to 0.5 and see how the response changes. The prompt is run again in the playground and this time the response is more detailed and more creative. The model has generated a response in three sections: “The Arrival”, “The Greeting,” and “First Contact.” Figure 1.7 shows excerpts from the response generated by the model. The response contains elements of surprise, fear, and shock from human beings around the world, and the model also narrates how the world came together to greet the aliens. The model further describes the first contact between aliens and humans and shows what the aliens looked like.

The Greeting

Global leaders, scientists, and military personnel quickly organized an international task force to approach the visitors. The United Nations had established protocols for such an event, though most had believed they would never need to be used. Representatives from various nations, equipped with translators, linguists, and advanced communication tools, were dispatched to the chosen landing sites.

In Central Park, a diverse group of diplomats, scientists, and security officers nervously awaited the first contact. A hastily erected stage and flags from multiple nations fluttered in the cold breeze. Around the world, billions of people tuned in to live broadcasts, their eyes fixed on the screens.

First Contact

A ramp extended slowly from the ship, touching down gently on the green expanse of Central Park. The air seemed to hum with anticipation. Then, from the ship emerged a delegation of extraterrestrial visitors. They were tall, slender beings with a luminescent quality to their skin, which seemed to pulse with a soft light. Their eyes, large and deep, conveyed an intelligence and curiosity that was palpable even across the vast cultural chasm that separated them from humanity.

Figure 1.7 The response from the model with the Temperature setting of 1 and Top P setting of 0.5.

Open AI, in addition to the settings we just modified to tune the response, provides batch processing of requests using the API endpoints. Developers can leverage these endpoints to get asynchronous responses with 50% lower costs than synchronous ones. Developers can also fine-tune the model for their use cases within their organizations or personal use using Open AI's API endpoints, custom training, and validation datasets. Open AI's extensive documentation for fine-tuning provides a treasure of resources for anyone looking to sharpen their prompt engineering skills, fine-tune the model for their use, and then generate responses.

1.8.2 LangChain

LangChain is an open-source framework that is available in Python and JavaScript. Developers and prompt engineering enthusiasts can leverage the framework to combine LLMs with other tools to build full-fledged LLM AI tools. One of the ways prompts can be made effective is to ensure context is maintained when creating prompts in subsequent conversations with the LLMs. LangChain ensures coherence, and the relevance of prompts is maintained during the creation process, which helps the LLMs generate effective responses. LangChain does this by providing developers a way to run experiments on datasets along with prompt templates that can be used to craft prompts that can then be sent via APIs to models for processing. Figure 1.8 shows the default code that LangChain provides for engineers to craft prompts for their unique datasets and desired models. LangChain also enables engineers to choose the different types of pre-built evaluation chains for common scenarios such as "Conciseness," "Relevance," "Coherence," "Harmfulness," etc. Engineers can choose the evaluation they would like to use against their experiments, and LangChain populates the base code with all default options along with placeholders for API keys, etc.

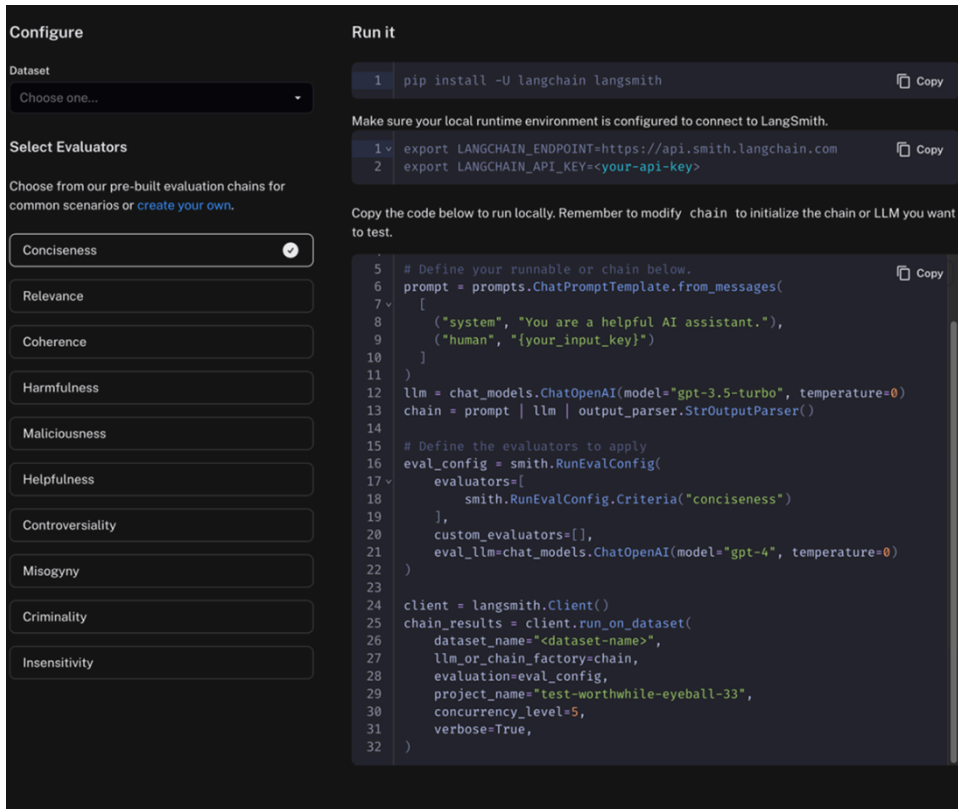


Figure 1.8 Engineers can use LangChain's web interface to choose an uploaded dataset and create a prompt experiment against the dataset and the desired model along with evaluations that can help evaluate the experiments.

LangChain also provides a Prompt Playground that users can use to run prompts against their desired models. This enables a one-stop shop for users to leverage all their models and prompts. Users can choose the supported provider, add the API keys, then use the default template or provide their template, create prompts, and test them against the model. The Prompt Playground, just like the Open AI Playground, provides more options to tune the model responses. In future chapters, we will go over them in detail when we talk about prompts, types of prompts, and how to leverage different model parameters to get desired outputs. Figure 1.9 shows the view of the Prompt Playground with a default prompt template with inputs and the output using the Open AI model gpt-4. The playground allows users to commit their prompts, look at the history of the prompt, etc.

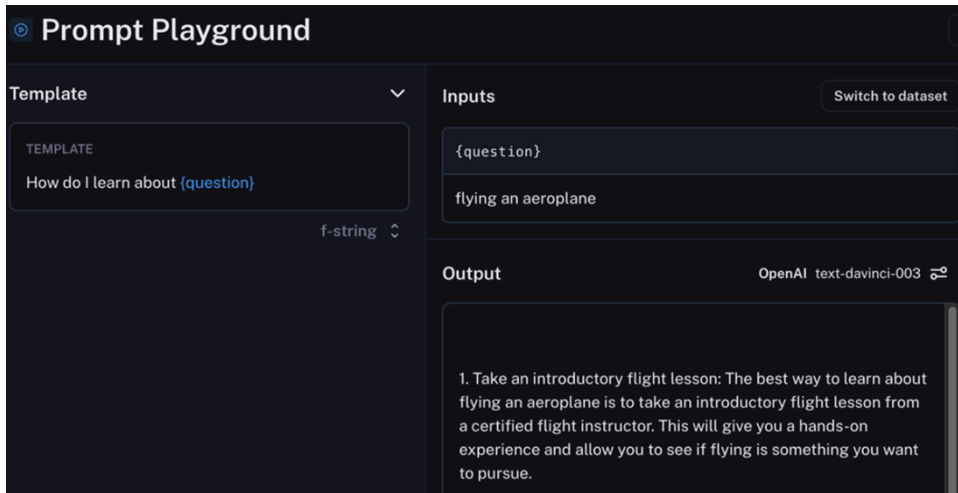


Figure 1.9 LangChain's Prompt Playground that allows users to use a template and craft prompts against models while maintaining history of the prompts used and the responses obtained.

Last is LangChain Hub, which provides a way to search all publicly available LangChain-supported prompts, models, etc. This helps users get up to speed sooner by leveraging prompt templates and types created by other users in the community. Therefore, LangChain provides many techniques to test prompts, craft effective prompts, evaluate prompts against models, and use these techniques to build LLM apps.

If you have followed through this far, you are ready to start crafting effective prompts using tools, models, and any datasets you may want to use. If you have not heard of any of the topics we have gone through so far, then don't worry. We will review each topic and many more in detail in the upcoming chapters.

1.9 Summary

- Large Language Models (LLMs) are now used widely across industries and disciplines by professionals, researchers, and students for content generation, modify, and interpret text, as well as produce meaningful insights.
- At the core of LLM usage is the concept of a prompt – an input that enables users to interact with the model and direct it to generate the specific response. Developing the skills of crafting effective prompts is essential for anyone who wants to maximize the capabilities of LLMs.
- LLMs are powerful but can produce hallucinations—factually incorrect or nonsensical responses. To mitigate these, it's important to use strategies such as, verifying outputs, iterative refinement, and feedback loops. Fine-tuning the model with domain-specific data also helps enhance accuracy and reduce over-generalization.

- Generative AI models include various types beyond text generation. For instance, text-to-image models like DALL-E create visuals, while text-to-audio models like Jukebox generate music and speech. Each model serves a specific purpose, showcasing the versatility of generative AI across creative, multimedia, and technical fields.
- Effective prompt engineering enables developers to ensure that the LLM is context-aware, ethically aligned, and capable of producing desired outputs iteratively.
- In this book, we will explore various prompting techniques, implement prompts programmatically, test them, and deploy them using tools such as LangChain.
- By the end, you will be equipped to craft effective prompts and develop, test, and deploy real-world applications with the latest generative AI tools.

2 Prompt Patterns: Basic Types and Templates

This chapter covers

- The fundamentals of prompt patterns with illustrated examples of using prompts with and without patterns.
- The overview of different prompt patterns, core elements, and relevance in prompt engineering.
- Different types of prompt patterns and their templates with examples of using prompt patterns to guide models to generate accurate and relevant responses.

Constructing effective prompts using prompt patterns and templates with real-world examples. In Chapter 1, we covered the fundamentals of prompt engineering, including the tools and technologies involved, as well as how prompt crafting differs among various language models (LLMs). In Chapter 2, we will build on these foundational concepts by exploring patterns and templates. These strategies are essential for creating more structured and effective prompts that elicit accurate and relevant responses from LLMs.

So, what does a pattern mean when it comes to prompts? At its core, a pattern is a structured approach or a predictable form that can be applied repeatedly to guide responses. Patterns consist of instructions and context intended to enhance accuracy and relevance when generating answers. These patterns serve as frameworks, allowing users to address specific types of tasks or questions with consistent results.

How do patterns work within prompts? Is it an additional field in a prompt? No, a prompt pattern is an intentional structure that the prompt follows to achieve specific objectives. This structured approach often relies on a template—a predefined format that guides the model on how to interpret the prompt. Templates are organized frameworks within patterns that include placeholders or guidelines, creating a consistent structure for the model to follow in response.

Beyond enhancing response quality, prompt patterns serve a practical purpose in prompt engineering. They allow users to select the right approach for specific problems, making it possible to generate consistent, reusable prompts across different tasks, datasets, and use cases. Let's now examine the components of a prompt pattern and how each element contributes to generating the desired outputs from a model.

2.1 Elements of a Prompt Pattern

A prompt pattern consists of several elements. The first set of elements are context, persona, tone, rules, limitations, and roles that guide the model in generating a response. The second set of elements is the prompt itself. Sometimes, you will notice that the all the elements are not fully discernable but form a pattern. These patterns tend to be verbose, allowing the model to have creative responses. Before we go deeper into each component that forms a prompt pattern, let us look into an example. In the following example, we provide the model with ten numbers and direct it to sort them from lowest to highest. The model responds with the sorted numbers in the requested order.

Here are ten numbers: [200, 1, -100, 90, 45, 23, 12, 8, 90, 100].

Please sort them from the lowest to the highest.

The response from the model is as follows:

Here are the ten numbers sorted from the lowest to the highest:

[-100, 1, 8, 12, 23, 45, 90, 90, 100, 200]

Notice something about the prompt? We provided some data before we prompted the model with instructions on what to do with the data. Let's break this down into the two sets of elements we discussed earlier. The first element consists of context, rules, limitations, and roles. In our example, the context is the list of the ten numbers, and the rules are in the second part, where we ask the model to sort the numbers from lowest to highest. The two elements combine to form what we call an *instruction-following prompt*. The components of an instruction-following prompt are:

- A set or list of data.

Clear and concise instructions on how to process the data. Similarly, different prompt patterns contain elements in various combinations necessary to guide the model in generating a desired output. Understanding what elements are mandatory in what type of pattern will help craft effective prompts. Let's look at another example. The following example shows a Game-Play Pattern used to play a simple game of guessing numbers.

Let's play a game where you ask me to guess a number between 1 and 100 and at the same time you randomly choose a number between 1 and 100. You ask me to enter the number and then match it with the number you sampled between 1 and 100. If the numbers match then I win else you win. Now, ask me to choose a number between 1 and 100.

The model responds with the first question in the game.

Alright, let's play the game!

I'm thinking of a number between 1 and 100. Can you guess what it is? Choose a number between 1 and 100 and let me know your guess.

Following the lead of the model, we try to guess the correct number

54

The model parses our response and checks if the number selected by us was correct.

Great! You guessed 54. Let me check if it matches my randomly selected number.

Randomly selecting a number...

The number I randomly selected is 23.

Sorry, the correct number was 23. I win! Would you like to play again?

Can you figure out what type of elements are forming the Game-Play prompt? Notice that the prompt contains the following:

- Instructions that guide the model in playing a game
- Rules to choose for both players to select a random number between 1 and 100
- A decision on whether the user won by matching the number selected by the model and the user.
- A prompt for the model to begin the game by asking us to choose between 1 and 100.

It will probably take some practice to discern all of the key elements clearly. In this case, the Game-Play pattern consists of four elements: context, rules, roles that define how to play the game, and a prompt that starts the game. The examples we saw so far illustrate how different patterns have different elements. The key to crafting effective prompts leveraging patterns is understanding the key elements required for each prompt pattern.

Having looked at two examples, we now have a mental model of how all or some elements can be used to construct an effective prompt. Figure 2.1 shows a sample prompt with color-coded elements of the prompt.

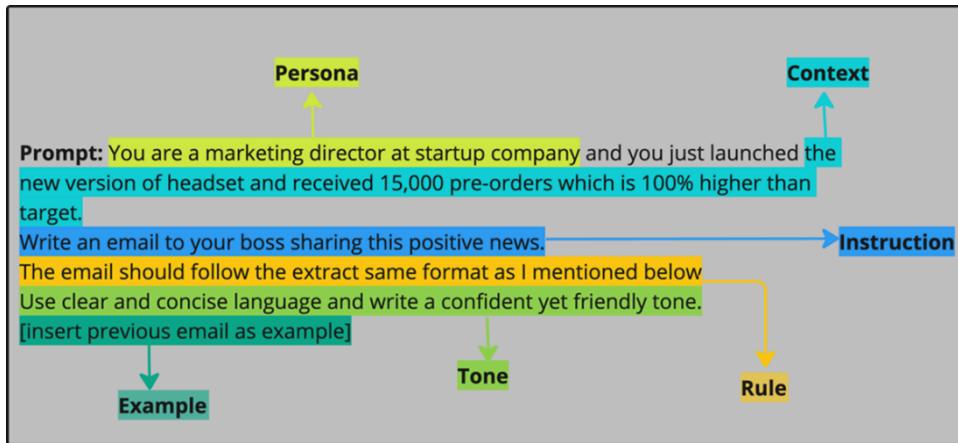


Figure 2.1 A sample prompt showing the elements of the prompt color coded.

Therefore, the elements in a prompt pattern can be defined as follows:

Persona

The persona in the pattern defines what roles the model and the user should assume during the interaction. This will help the model understand the limitations of responses, the type of responses, and the level of detail of the responses.

Tone

When you construct a prompt, you can guide the model's response to reflect the tonality that you feel is appropriate. This element is essential when working with the model to generate reusable templates, convey happy, sad, and informative messages to your audience, improve a text to sound more engaging, etc.

Rules

In most cases you want the model to abide by rules you set out to get the desired output. It would be best if you remembered to include these rules as a part of the prompt. Prompt patterns enforce the inclusion of rules by including them in the pattern wherever applicable.

Limitations

In use cases where a certain level of randomness and creativity is required from the model, it is essential to include them as limitations in the prompt. These limitations guide the model in generating random responses for the same or similar prompt. You could use this to make a chatbot that greets users slightly differently every time giving it a more human touch. You can also use limitations to specify the extent to which you would like the data to be processed or the response to be creative thereby controlling the response when the model is used at a large scale.

Input Data

The input data defines what the model should use to process the prompt. You can include the input data in various prompt parts depending on the pattern type. The data could be a text file, image, etc., depending on the model's capability.

Prompt

The prompt is the element that guides the model on how to interpret the input data, tone, rules, limitations, and personas before generating the response. Prompt patterns provide a template for prompts that can be reused for your use cases.

ARE ALL ELEMENTS ESSENTIAL? All the elements we discussed so far are present in only some patterns. Depending on the scenario, you can construct prompts with a subset of the elements. However, these elements are the most common and will be essential for complex use cases.

THE TYPE OF LLM COUNTS! The effectiveness of a prompt pattern can vary based on the type of Large Language Model (LLM) being used. For example, an LLM fine-tuned for mathematics may require fewer rules and instructions to generate accurate responses, whereas a general-purpose model may need more context, role definitions, and detailed instructions to adapt the same pattern. Selecting the right model is crucial for maximizing the effectiveness of prompt patterns.

A unique characteristic of large language models (LLMs) that can affect their outcomes is their stochastic nature. This means that the same prompt may yield different responses each time it is used. This variability can be managed with prompt stability along with reusable prompt patterns, such as persona, question-refinement, and recipe patterns. Let's explore the stochastic nature of LLMs that helps to understand the importance of prompt patterns.

2.2 Stochastic Responses and Prompt Stability

Large Language Models (LLMs) generate responses based on probabilities, which means that the same prompt can generate different answers each time it is used. This variability is due to the stochastic nature of LLMs, as these models randomly choose from a range of plausible responses. The selection is influenced by parameters such as temperature and top-p, which affect the diversity and creativity of the responses.

STOCHASTIC NATURE OF LLM RESPONSES

The unpredictable nature of large language models (LLMs) can be beneficial because it fosters creativity and helps avoid repetitive or overly deterministic responses. However, this means that when you enter the same prompt multiple times, the responses may vary slightly in wording, detail, or emphasis. This variability is particularly noticeable when the model is directed to be more creative, such as when a higher temperature setting is used.

PROMPT STABILITY

Prompt stability refers to the consistency in the quality of responses generated by the model when the same prompt is used multiple times. Some prompts produce stable and high-quality responses consistently, while others may result in a wide range of response qualities. This variation can be controlled by prompt engineering and adjusting parameters such as temperature. Lowering the temperature can reduce randomness, making the responses more predictable and enhancing stability. To improve response consistency, try:

Refining Prompt Specificity

Adding specific instructions, examples, or context can narrow the scope of responses, reducing randomness.

Using Lower Temperature

Adjusting temperature settings to a lower value (e.g., 0.2) encourages the model to produce responses that are less creative but more consistent.

Iterating and Selecting

Run the same prompt multiple times, then select or refine the best responses based on stability and relevance.

By effectively managing stochastic responses and ensuring prompt stability, users can improve the quality and consistency of outputs generated by LLMs. This is particularly valuable in applications that demand reliability and precision.

Consider when the clarity and effectiveness of a prompt are compromised by unnecessary information or poor structure, it can result in less accurate or irrelevant responses from language models. This phenomenon is known as prompt dilution. Prompt dilution can be reduced with prompt patterns, which are structured frameworks or templates designed to create clear and effective prompts.

Alternatively, multi-agent systems can avoid prompt dilution, which is achieved through API integrations. These systems employ multiple agents, each with its own specific instructions, tools, and integrations. This specialization ensures that each sub-agent is trained and optimized to handle particular types of requests, thereby reducing the likelihood of errors. By focusing on relevant information pertaining to their specific tasks, each sub-agent minimizes confusion and decreases the chances of generating incorrect or irrelevant responses. Now, let's explore how multi-agent systems operate.

2.3 Multi-Agent Systems

Multi-Agent Systems consist of autonomous entities, known as agents, that interact within an environment to achieve individual or common problems. These agents can communicate, collaborate, or compete and are utilized in various fields such as robotics, distributed computing, artificial intelligence, and economics.

WHAT IS AN AGENT?

Agents in prompt engineering refer to autonomous processes or models capable of dynamically determining actions based on specific inputs. They adapt workflows by selecting steps in real-time, enabling handling complex and evolving tasks.

Key features of Multi-Agent Systems:

- **Autonomy:** Agents operate without direct human involvement.
- **Decentralization:** Agents have no single point of control.
- **Coordination:** Agents work together to solve complex problems.
- **Communication:** Agents share information to improve decision-making.

Customer: "Where is my order?"

Main Agent: Routes query to order tracking sub-agent using intent-based routing.

Order Tracking Sub-Agent: "Hello, I can help you track your order. Please provide your order number or tracking number."

We learned how to handle the queries if the clarity and effectiveness of the prompt are reduced through agents; let's delve into specific types of patterns. Each pattern type addresses unique use cases, offering strategies to manage variability and ensure consistent results. In the following section, let's explore the various types of prompt patterns and their applications through examples.

2.4 Types of Prompt Patterns and their templates

So far, we have seen the elements of a prompt. All or some of the elements we discussed combine to form a prompt pattern. This section will examine the basic and commonly used prompt patterns and their templates. Each pattern we discuss has a template defining the essential elements required to craft an effective prompt. You will encounter patterns where a particular element plays an important role. For example, the persona pattern's essential element is the persona element. While other patterns may or may not use the persona element, the persona pattern uses this element to define the model's persona. The model, assuming the role defined by the persona, generates responses for prompts crafted following the persona pattern. Therefore, this pattern is called the Persona pattern. Similarly, we have the Audience Persona pattern, where the persona is the core element, and it defines the audience's persona.

Prompt patterns, based on their use cases, are classified into different categories. Figure 2.2 shows the categorization of the basic prompt patterns that we are going to talk about.



Figure 2.2 shows the basic prompts categorized by their use cases.

Figure 2.2 helps is important to understand where a particular prompt pattern can be used. Now, let's look at each of the patterns in detail.

2.5 Output Customization

The response generated from large language models (LLMs) can vary across different iterations and models, making it challenging to build LLM applications. You can use output customization patterns within the input prompt to ensure consistent responses in a particular format. Incorporating these patterns can be especially helpful when creating content-generation chatbots. Here is a simple example of an output customization pattern.

Act as a mathematics tutor, generate a concise and simple plan in bullet point outline on the topic of algebra.

Let's explore various output customization patterns and observe how these patterns enhance the LLM-generated responses.

2.5.1 Persona Pattern

A Persona pattern enables you to assign a specific persona to the model when generating responses. The model adopts the expertise or perceived expertise of the persona and tunes the responses to it. With the persona pattern, you can enable the model to generate responses tailored to specific user groups. This pattern is particularly useful where you want the model to create consistent, relevant, and predictable responses. To help understand this better, assume you have a math question you are struggling with. If you need help with the question, you reach out to an expert in math or refer to a resource that can give you hints to solve the question. Similarly, models can assume the persona you choose and create responses based on the persona. Imagine you have a product marketing problem that you would like answers to. You can use the persona pattern to model your prompt in a way that guides the model to assume the role of a product marketing expert and then answer your question. The transcript that follows shows how to direct the model to assume the role of a travel guide. The model's response shows that it has taken this role. Further questions to the prompt will generate responses like you are interacting with a travel expert.

Act as a travel guide. When I ask you a question please respond in detail.

Absolutely, I'd be delighted to act as your travel guide. Ask me any travel-related questions, and I'll provide detailed responses to help you plan your trip.

In the next transcript, we utilize a persona pattern to shape the model's responses by defining it as a specific system role—essentially, creating a job description for the role we want the model to fulfill. A well-defined persona serves as a guide, much like writing a job description for the type of "assistant" you would hire. It outlines their areas of expertise, the tone they should adopt, and the types of tasks they are expected to perform. By clarifying these expectations, the model can generate responses that are more consistent, relevant, and tailored to the intended audience.

Act as a marketing expert. I will provide product details, and you will respond with a detailed marketing plan to improve the product's reach

Absolutely, I'm ready to assist with your marketing strategy. Please provide the details of your product, including its features, target audience, unique selling points, current market presence (if any), and any specific goals or challenges you're facing. With this information, I can tailor a comprehensive marketing plan to enhance the reach and success of your product. Let's get started!

Hand-made gender neutral backpack designed to fit a laptop and accessories for working professionals.

Great! A handmade, gender-neutral backpack designed to accommodate laptops and accessories for working professionals has significant potential in the market. Here's a detailed marketing plan to improve its reach:

Define Target Audience: Identify the primary demographic for the backpack, such as working professionals aged 25-45 who value functionality, style, and sustainability.

Create Compelling Brand Story: Develop a narrative around the brand's ethos, craftsmanship, and commitment to meeting the needs of modern professionals while promoting gender inclusivity and sustainability.

We have not included the complete response for brevity, but you can try this on your own and see how the model responds. Having looked at a few examples, let's now analyze what key elements can help craft an effective persona pattern for the model to generate coherent, consistent, and optimized responses. Figure 2.3 shows the workflow for creating a prompt using the critical element, persona, and adding the context and instruction. The workflow acts as a pattern that can be reused to craft various persona prompts.



Figure 2.3 shows how the key elements form a Persona pattern.

We have now seen a basic example of creating a persona prompt using the key elements: Persona, Context, and Instruction. However, for complex use cases, you will craft the prompt and refine the prompt iteratively until you get the desired response from the model. So, let's define the workflow of creating a persona prompt using the persona pattern.

Defining a Persona

Define the persona you wish an LLM model to embody and generate responses accordingly. You can craft this persona in alignment with the specific problem statement you aim to address, ensuring tailored responses. For example, a persona could represent a tax consultant, a mathematician, a medical advisor, or a domain expert.

Phrasing the Prompt

Crafting the prompt with the chosen persona and providing proper context and details about the specific question yields a tailored response. For instance, if you select "tax consultant" as the persona, the context should be "analyze tax documents and provide tax-saving tips based on salary and occupation."

Guiding the Response

The persona pattern influences not only the response's style but also its content and format. You can embed contextual information in the persona to guide the model in creating responses in any form, such as creative, professional, or casual, which can be embedded with the chosen persona.

Prompt-Tuning

Adding additional context or constraints to the prompt will fine-tune the persona, generating the optimized response. For example, you can instruct the LLM to respond as a “tax expert specialized in tax savings” for an optimized and coherent response.

Feedback Loop

Based on the initial response from the first prompt, understand the response and refine the prompt or persona to improve the quality and consistency of the LLM-generated content.

The Persona Pattern is a simple yet powerful technique that enables users to emulate real-world scenarios, resulting in tailored responses from LLM models. To craft the prompt with a persona pattern, follow the steps mentioned in the following section.

FORMAT OF THE PERSONA PATTERN

Act as Persona X. Perform task Y.

In addition to the format, you can apply additional contextual statements while designing the prompt to guide the model.

You will need to replace "X" with an appropriate persona, such as "travel guide" or "scrum master," and then specify a problem or task to perform. Coming next, we'll see a similar pattern, but with a key difference. Rather than telling the model what persona it should be, we will be telling the model what persona it should have in mind when generating its response.

2.5.2 Audience Persona Pattern

The Audience Persona Pattern involves creating responses tailored to a specific audience's characteristics and preferences. This pattern differs from the Persona pattern, which involves asking questions as fictional personas. Unlike the Persona pattern, the Audience Persona pattern focuses on customizing content or responses to match the characteristics of the persona you assume. Imagine you want to understand a programming language with a fifth-grade mindset, but you prompt the model with the prompt shown next.

Explain the core concepts of programming language

The core concepts of programming languages are fundamental principles and components that underlie all programming languages. These concepts are crucial for understanding how programming languages work and for effectively writing code.

Notice the response is generic and this is because the prompt was not tailored to the desired response. Let's now attempt to add more context and create an effective audience persona pattern prompt. The following transcript shows the improved pattern with the response from the model.

Explain the core concepts of programming language. Assume I am a fifth-grade student.

Of course! Let's explain the core concepts of programming languages in a simple way for a fifth-grade student:

Now, the model responds with a tuned, relevant, and coherent response. We have not included the entire response for brevity. So, what did we learn from the examples? The core idea of crafting an audience persona pattern is to leverage the persona element, add context by using the audience's needs, personalize the prompt, and guide the model by including examples that suit the audience defined earlier. Figure 2.4 shows how a prompt using the audience persona prompt can be created.

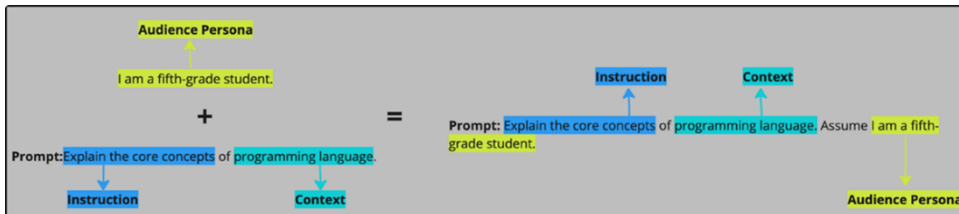


Figure 2.4 shows the creation of a pattern using the audience persona pattern.

Let's now look at the workflow you can use to craft a prompt using the audience persona pattern.

Identify the Audience

Define the target audience you wish an LLM model to act. By understanding the demographics and characteristics, the audience could be domain experts, general users, professionals from specific domains, or specific age groups. Based on the above example, fifth-grade students are the target audience.

Audience Needs

Gather the audience's needs and insights based on their pain points, interests, and clarifications. This will help you to craft relevant and engaging prompts.

Audience-Specific Language

Match the tone and vocabulary of your prompt so that the audience is comfortable.

Audience Concerns

Specify the concerns or biases while designing the prompt; this will help gain confidence and trust in building the response from the LLM.

Personalize Prompt

Mention the audience type or name based on previous conversations with the language model. This personalization makes the conversation even more engaging.

Relevant Examples

Embed the examples within the prompt that suit the audience domain or industry, which helps to clarify the content appropriately.

FORMAT OF THE AUDIENCE PERSONA PATTERN

Explain X to me. Assume that I am Persona Y.

You will also need to specify the topic X and you will need to replace "Y" with an appropriate persona, such as "An undergrad student" or "A five-year-old".

2.6 Prompt Improvement

Large language models have a wide range of applications and can enhance productivity in various fields. However, using these models effectively can be difficult without knowing how to create prompts. Crafting prompts is crucial for clear communication and generating optimal responses, and this requires knowledge of prompt improvement patterns. This challenge arises because only a few people understand how to construct effective prompts for communication with large language models. Let's now take a look at two patterns that can help you improve your prompts.

2.6.1 Question Refinement Pattern

The Question Refinement Pattern is a logical approach aimed at enhancing the quality of questions to extract coherent and appropriate responses. This method simplifies interactions with large language models by refining questions and aligning them more closely with problem statements, thus improving overall conversation quality. The fundamental idea behind this pattern is that we may need precise information or context specific to the question to answer it. Models can, however, frame the question with additional context provided we craft a prompt to guide them.

Imagine you are planning to go out for dinner and want to try the best Italian food. If you prompt the model with the question, "What's the best Italian food?" the response may be too generic. You could, however, leverage the model to help refine the question. The following example shows how to use the model to refine your questions.

Whenever I ask a question, suggest a better question and ask me if I would like to use it instead.

The model responds with the following text:

Understood! I'll provide a suggested improved question and ask if you would like to use it instead in response to your queries. If you have a question you'd like to start with, please go ahead, and I'll provide a suggestion based on that question.

We then ask the model about the best Italian food:

What's the best Italian food?

The model based on the earlier guidance now refines the question.

Certainly! Here's a refined question to help you get more specific recommendations:

"Could you please provide recommendations for the best Italian dishes to try? I'm particularly interested in [specific type of Italian cuisine or region] cuisine."

This refined question allows you to focus on specific Italian dishes or regions of interest, which can lead to more tailored recommendations. Would you like to use this refined question?

Having seen the example, we can create a prompt using the question refinement pattern using the following steps:

Initial Prompt: Initialize the starting prompt to suggest a version of a question whenever we ask a fundamental question without additional context. Here is an example:

Whenever I ask a question, suggest a better version of the question to use instead.

Start with Basic Question: Begin with fundamental questions about your topic with or without additional context.

Evaluate the Response: Evaluate the refined question generated by a large language model and add context to subsequent questions if needed.

Iterate and Refine: Based on the response received, further refine your question to delve into more specific areas or to clarify any misunderstandings.

FORMAT OF THE QUESTION REFINEMENT PATTERN

When I ask X question, suggest a better version with Y constraints. You need to replace X in the format, Y with the constraints within which the model should generate the improved version of X.

2.7 Interaction

The primary goal of interaction patterns is to improve or optimize interactions with models and encourage them to actively engage in conversations instead of being passive. These patterns are particularly useful for creative applications like knowledge testing across different subjects, voice chatbots, and gaming.

2.7.1 Game-Play Pattern

As the name suggests, the Game-Play pattern is about creating a game-like interaction with the LLM. It involves creating a text-based game by defining its rules and roles around a specific topic and then engaging in a dynamic play with the LLM. The game is composed of two main elements: rules and content. The rules set the game's framework and define the roles of the participants (user and the LLM), while the content provides the context for the gameplay, which the LLM will use to devise the game.

The underlying concept of this pattern is to guide the LLM in generating scenarios and questions related to a specific topic and enhance your knowledge or skills in that area by answering the questions or scenarios using logical reasoning or problem-solving skills.

The practical applications of the Game-Play pattern are extensive. Consider a scenario where you want to enhance your proficiency in a specific topic. You can leverage the Game Play pattern to test your knowledge of the subject. Taking it a step further, if a company has revamped its security training and wants its employees to undergo the training and then take a test, they can employ the Game Play pattern. This would enable the LLM to generate questions and scenarios based on the training material.

In the field of education, this pattern can be a game-changer if implemented on a large scale. Students, teachers, and teaching assistants can use this pattern to create quizzes and personalized learning paths for various subjects like mathematics, reading comprehension, physics, and more. Students can use this pattern to assess their skills, while teachers can use it to evaluate the students' knowledge, ensuring that the questions are not repetitive and have a more engaging structure. The following example shows a sample game-based prompt.

Let's have some fun with prompt engineering. Think of a simple task I can perform following your prompts. Your tasks must involve logic or programming skills, but you do not need to write code.

My task is to create a prompt that solves a problem. You will test any prompt by giving me the output and evaluating how well it worked. I will ask you to stop when I no longer want to play. Please ask me one question at a time and wait for my answer before asking another one. After that, evaluate my response.

Now, ask me to create a prompt.

The model starts the game based on our instructions:

Got it!

Task 1: Sorting a List of Numbers

Question: Please create a prompt that sorts the following list of numbers in ascending order: [34, 7, 23, 32, 5, 62].

Please sort the given numbers [34, 7, 23, 32, 5, 62]

Great! Here is an evaluation of your response.

Evaluation of Your Prompt:

Your Prompt: Please sort the given numbers [34, 7, 23, 32, 5, 62].

Output:

- Sorted List: [5, 7, 23, 32, 34, 62]

Evaluation:

- Clarity: The prompt is clear and direct, specifying exactly what needs to be done.
- Effectiveness: The output accurately reflects the sorted order of the numbers provided.
- Succinctness: The prompt is concise, avoiding any unnecessary information.

The prompt successfully achieved the goal of sorting the list of numbers. Well done!

Would you like to continue with another task?

The example we saw with ChatGPT shows that the Game-Play pattern has a format that can be repeated for many use cases. Let's now look at the format to create an effective prompt. Figure 2.5 shows how you could construct a game-play prompt using the context, instruction, and role definition, which are part of the game-play.

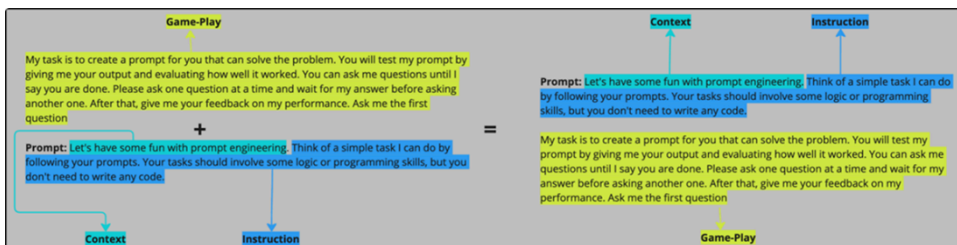


Figure 2.5 shows the construction of a game-play prompt following the game-play pattern.

FORMAT OF THE GAME-PLAY PATTERN

Create a game/Let's play a game based on topic X, keeping in mind Y fundamental rules, and Z roles, E contexts.

You can replace X with your desired topic when creating a prompt based on the gameplay pattern, Y with the fundamental rules, Z with the role definition for you and the model, and E as any other contextual information required to clarify the game.

2.7.2 Template Pattern

The Template Pattern enables users to create a structured format or template that the language model should adhere to when generating responses. This method is beneficial when consistent output formatting is crucial. For instance, you can utilize the Template Pattern to write articles, organize responses in specified formats, or transform text from one structured format to another by providing the appropriate template. The following example shows a sample use case for the template pattern.

Template: Question: [Interview question] Answer: [Answer to question]

"Please generate a set of interview questions for prompt engineering following the template."

Based on our instructions, the model generates the following set of questions:

Sure! Here is a set of interview questions for prompt engineering following the given template:

Question: How do you ensure clarity in a prompt when instructing an AI model to perform a task?

Answer: Ensuring clarity in a prompt involves using precise language, avoiding ambiguity, and providing enough context for the AI model to understand the task. It is also important to use specific examples or formats when necessary to guide the AI.

Question: What strategies can be used to improve the effectiveness of a prompt?

Answer: Strategies to improve the effectiveness of a prompt include testing the prompt with different variations, analyzing the outputs to identify issues, iterating on the prompt based on feedback, and ensuring it aligns with the desired outcome. Additionally, keeping the prompt focused and concise helps improve its effectiveness.

A key aspect of the template pattern is that it limits the LLM's creativity because we are providing a rigid template that the LLM should follow. However, the template ensures consistency and predictability in how the model should respond to your prompts. You can use the template pattern to set up a template for a creative story, for question and answers, for text summarization, and more. At the heart of a template pattern is defining the correct template. The format of the template pattern consists of the following:

- **Template:** Create a template that defines the format of the expected response. The template may contain slots for the model to understand and fill in with generated responses.
- **Context:** Provide context around what the template is about. The context could be an input file, data, etc., followed by information on processing or understanding the data.
- **Prompt:** Guide the model in adhering to the template and following the context to answer the prompt.

The template pattern, when effectively crafted, would look like the following:

Template: Introduction: [Introduction]. Section 1: [Key points summarized in Section 1]. Section 2: [Key points in section 2]. Conclusion: [Closing thoughts]

Prompt: Please generate an outline for an article discussing the benefits of prompt engineering.

Figure 2.6 shows the construction of a template prompt using a template with placeholders for “interview question” and “answer to the question.” The model parses the prompt, context, and template and generates sample questions and answers following the prompt.

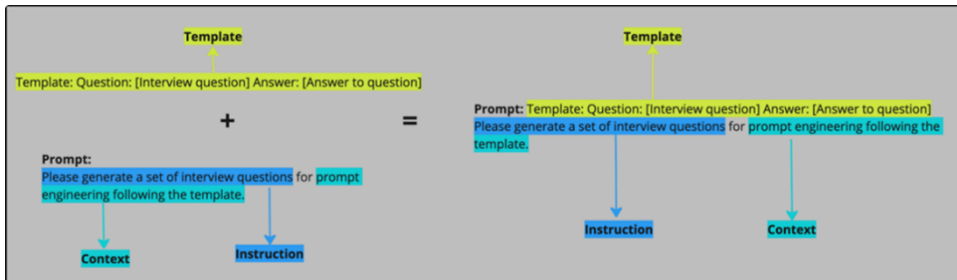


Figure 2.6 shows an interview question and answer generation prompt leveraging a template.

FORMAT OF THE TEMPLATE PATTERN

Template: <User defined text>: [Placeholder for the model to fill]

Prompt: Create/Generate/Fill in the blanks based on X.

When creating the template pattern, you should lay out the template in as much detail as possible, ensuring the model can understand it and create a response matching it. The prompt related to the template should also have a context that can guide the model in developing responses that match the template and the context.

2.8 Input Semantics

By now, we understand the power of language models in extracting information based on the context and instruction in the form prompt. However, with input, semantic patterns help to create another user-defined language, such as shorthand notation. This pattern is handy in communication-intensive jobs such as call centers. Let’s now take a look at the Meta Language Creation pattern.

2.8.1 Meta Language Creation Pattern

Meta Language Creation is about creating a shorthand to communicate with a model when speed and brevity are required. This pattern enables users to guide the model in following special linguistic rules that are valid in a specific context. You can use this pattern to guide the model to a certain behavior without having to specify the behavior again if the context is maintained. For example, you want to ensure that every time you use the text "T2EF" the model should translate the following text into Spanish and French. You can guide the model to parse T2EF to trigger a translation, and the model will translate the text into Spanish and French every time you use T2EF in the same context. Let us look at an example before we understand this pattern in detail. First, we will see how to create a meta-language pattern, and then how to use the meta-language prompt to generate the desired response.

When I say "2P12NS", you should understand that as a rule to generate 2 random passwords of 12 characters, including numbers and special characters.

Got it! When you say "2P12NS", it means to generate 2 random passwords of 12 characters each, including numbers and special characters.

Now, let's prompt the model with the pattern "2P12NS".

2P12NS

The model parses the prompt and generates the following response:

Understood. Following the rule "2P12NS", here are two randomly generated passwords of 12 characters each, including numbers and special characters:

1. &7H3\$5f@9G1%
2. *2k@6#8L9p\$4

Would you like to continue with another task or have any other requests?

Figure 2.7 shows the creation of a meta language prompt using the elements: meta-language, instruction, and context. You can use this pattern to create more meta-language prompts for triggering analysis, translation, etc. whenever the meta text is encountered.

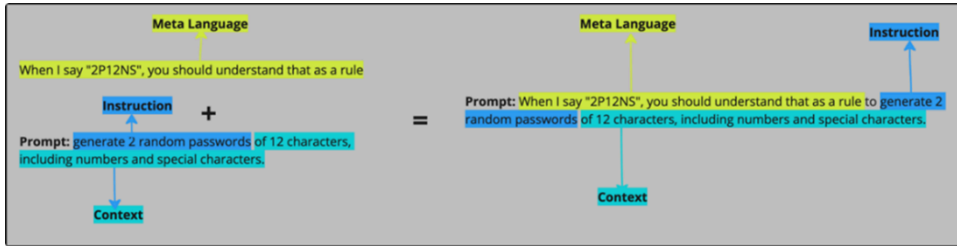


Figure 2.7 shows how the meta language prompt is created.

FORMAT OF THE META LANGUAGE CREATION PATTERN

When I say [X], I mean or would like you to do [Y]. In this template, you should replace X with the original content and Y with how you would like the model to interpret the content in X.

2.8.2 Recipe Pattern

As the name suggests, this pattern allows you to create a recipe of ideas and steps to accomplish a task. Using this pattern, you provide the model with partial information and context. Using the information provided, the model will fill in the gaps to create a narrative or analysis. Imagine that you want to write a short story, and you have a rough idea of the characters, their roles, and the high-level direction of the story. You can use this pattern to create a recipe of steps and ask the model to write the story creatively.

Task Description:

Generate a creative story based on a given prompt

Input Format:

Prompt: A single sentence prompt that provides the theme or idea for the story.

Max Length: 600 words.

Instructions for Interaction:

Step 1: Begin with the provided prompt.

Step 2: Ensure the story is coherent and engaging.

Step 3: Generate a story with a maximum length of 600 words. If the story exceeds 600 words then trim it.

Example Input and Expected Output:

Input: "The space shuttle took off with the 3 astronauts."

Output: "They were the last promise Earth had. They had to be successful for humanity to survive"

Evaluation Criteria:

Coherence: The story should have a logical flow.

Creativity: The story should be imaginative and engaging.

Optional Parameters:

Temperature: Set temperature to 0.8 for creativity.

For the sake of brevity, we haven't pasted the entire result containing 600 words. However, feel free to try the prompt on a couple of different models and see how they respond. In our experience, Copilot has had the best response so far. Notice any elements of the prompt in this example? The prompt has a task description that guides the model on parsing the next set of instructions. This input format consists of the prompt with some limitations, instructions on how to start the interaction, sample inputs and outputs, evaluation criteria, and optional parameters. These elements help guide the model in generating a story in our example but can be used to elaborate on a sequence of steps comprehensively. At its core also lies a reusable format that can create similar recipes across different topics and specializations. Can you recognize it? Hint: the format is a part of the example. Figure 2.8 is a good example on how you can create a recipe prompt. Keep in mind this looks similar to the template pattern but the key difference here is that a recipe pattern will generate a detailed response from a recipe whereas a template pattern just fills the template with the queried data.

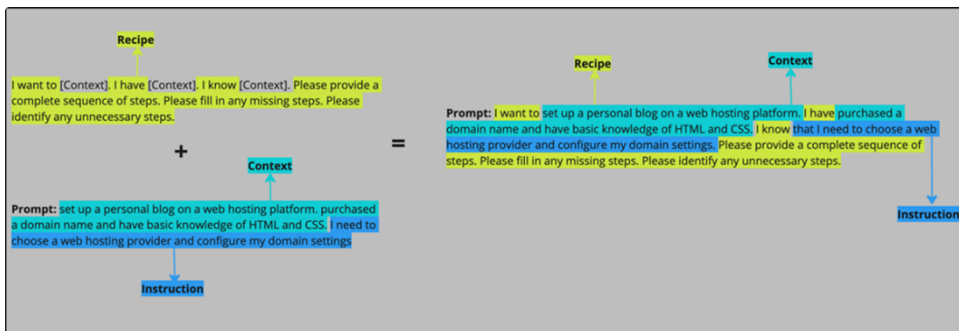


Figure 2.8 shows the creation of a recipe prompt to create a personal blog.

FORMAT OF THE RECIPE PATTERN

I would like to perform a task X.

The input would look like [A, B, C....]

The instructions on how to parse the input are [D, E, F....]

The sample input and output are [G, H]

The evaluation criteria is [The generated content must be ____, ____]

Optional context, parameters are as follows.

Although complex, this pattern can generate creative, comprehensive, and detailed responses across a vast range of topics and scenarios. You can use the format to create responses for many use cases. Let's take another example. Imagine your team is responsible for ensuring customers experiencing trouble due to shipment delays are duly compensated for their inconvenience. By using the recipe pattern, you could increase your team's productivity. The following example shows a step-by-step process to create a reusable template using the recipe pattern.

Title: Generating a Friendly Email Response

Goal: Generate a friendly and professional email response to a customer inquiry about a delayed order.

Ingredients:

- Tone: Friendly, Professional
- Style: Concise, Apologetic, Reassuring
- Key Information:
- Acknowledge the delay
- Apologize for the inconvenience
- Provide an update on the order status
- Offer a solution or compensation if applicable
- Thank the customer for their patience

The model based on our prompt generates the following response:

You are a customer service representative for an online store. A customer has written to inquire about the status of their delayed order. Write a friendly and professional email response. Make sure to acknowledge the delay, apologize for the inconvenience, provide an update on the order status, offer a solution or compensation if applicable, and thank the customer for their patience.

Subject: Update on Your Order #{{order_number}}

Dear {{customer_name}},

Thank you for reaching out regarding your order #{{order_number}}.

I sincerely apologize for the delay in your order. We understand how frustrating this can be and are doing everything we can to expedite the process. Currently, your order is [provide current status of the order, e.g., "being prepared for shipment" or "in transit"].

As a token of our appreciation for your patience, we would like to offer you [mention any compensation, e.g., "a 10% discount on your next purchase" or "free expedited shipping on this order"].

We expect your order to arrive by [provide an estimated delivery date].

Thank you for your understanding and patience. If you have any further questions or concerns, please don't hesitate to contact us.

Best regards,

{{your_name}}

Customer Service Representative

[Your Company Name]

Isn't it amazing that the model can create a reusable template based on the recipe prompt for building an email to respond to customers? Try the same example by changing some parameters or using a different scenario.

In this example, the Recipe Pattern breaks a task into sub-tasks, providing a step-by-step sequence for the LLM to follow. Each sub-task guides the model to address specific parts of the problem, ensuring the final output is comprehensive and structured. These problems can be simplified with agents, which will be covered in later chapters.

COMPARISON WITH AGENTS

In prompt engineering, agents refer to autonomous processes or models that can determine actions based on specific inputs. These agents are designed to handle complex tasks by dynamically selecting steps and managing workflows across different tools. In contrast, the Recipe Pattern consists of sub-tasks that act as fixed steps, guiding the language model (LLM) along a predefined sequence without the need for real-time decision-making. While agents are adaptable and capable of selecting actions based on changing inputs, the sub-tasks in the Recipe Pattern are set in advance. This offers a simpler yet effective way to handle structured tasks within a prompt. By using the Recipe Pattern, you can clearly outline each required step, ensuring that the model addresses all necessary components.

2.8.3 Alternative Approaches Pattern

This pattern encourages the model to offer different solutions, answers, opinions, or responses and allows you to choose the best solution for your problem. This pattern is very useful when you are not sure of the path ahead and want to explore several strategies that help you understand the best path forward for any given problem. Figure 2.9 shows how you can create an alternative approach prompt using the pattern of adding the context, instruction, and the alternative approach request.

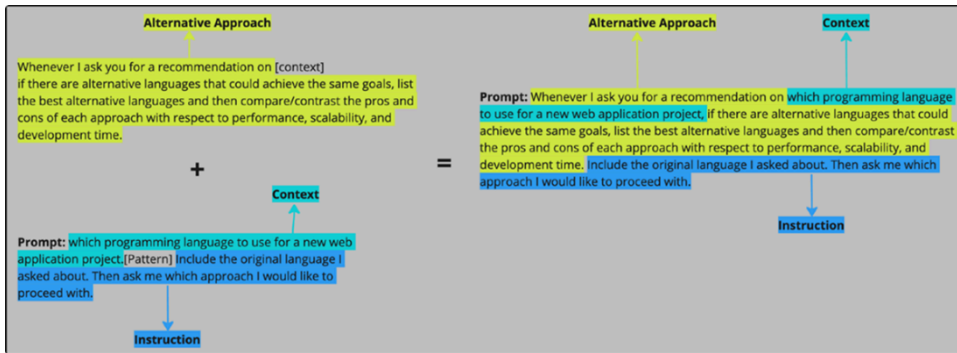


Figure 2.9 shows the creation of an alternative approach prompt.

Imagine you would like to improve your craft of crafting prompts using the model. How would you go about it? You need to let somehow the model know that while you are prompting it, it should respond with a better version of the model. If you get the model to do this, you can learn better strategies for creating effective prompts. The following example shows using the alternative approaches pattern we saw in Figure 2.9 to help you with finding some languages suited for a web development project.

Whenever I ask you for a recommendation on which programming language to use for a new web application project, if there are alternative languages that could achieve the same goals, list the best alternative languages and then compare/contrast the pros and cons of each approach with respect to performance, scalability, and development time. Include the original language I asked about. Then ask me which approach I would like to proceed with.

Can you recommend a few programming languages to use for a web development project?

The model responds with the following response:

Here are some programming languages recommended for web development, along with their pros and cons regarding performance, scalability, and development time:

1. Python (Django/Flask)

Pros:

Ease of Learning and Use: Clear, readable syntax; beginner-friendly.

Frameworks: Django and Flask offer comprehensive tools for web development.

Community Support: Large community with abundant resources and libraries.

Rapid Development: Quick prototyping and development cycle.

.....

Next Steps

Which approach would you like to proceed with? Let me know if you have any specific requirements or preferences for your project, and I can help narrow down the best choice for you.

In this example, we have yet to include the full response as it is very detailed and shows 4 languages with their pros and cons. This seems like a straightforward pattern from the example, but you can use this for many complex scenarios. Imagine you want to know how to approach the marketing plan for your company's product. You can use this pattern to guide the model in generating multiple options along with their pros and cons and then choose the one that best suits your needs. Now that we have seen examples and understand how to use this pattern let's look at its format.

FORMAT OF THE ALTERNATIVE APPROACHES PATTERN

From now on, when I ask X, suggest a better version/options/approaches.

You can replace X with a topic, prompt, question, etc. and the model will use this pattern to suggest alternative approaches.

So far, we have seen the elements of a prompt pattern, the way each of those elements play a role in creating different prompts, and we also saw in detail the basic and most used prompt patterns. Additionally, we saw the formats of the prompt patterns, and how to use them to create effective prompts. In the next chapter, we will look at advanced prompt patterns that are not very common but are powerful and have very diverse use cases.

2.9 Summary

- Prompt patterns are essential tools in crafting structured and effective prompts that guide LLMs to generate consistent and relevant responses. By breaking down prompts into key elements such as context, rules, limitations, and roles, users can create reusable patterns tailored to various tasks and applications.
- While prompt patterns can improve response accuracy, the inherent randomness of large language model (LLM) outputs may differ even with the same prompt. This variability is influenced by factors like temperature and top-p settings, which can introduce both creativity and unpredictability. To achieve reliable outcomes, prompt stability—is defined as the ability to generate consistent, high-quality responses across multiple attempts. Techniques such as refining prompt specificity, reducing randomness, and conducting iterative testing can help mitigate these effects.

- Persona, Audience persona pattern help mimic interactions with different personas and tailor responses based on the audience type chosen in the pattern. These patterns are useful in understanding the use cases of a varied audience/persona group.
- Question refinement pattern helps refine questions using the model. The model can add context and create templates where you can add context to refine and make relevant and coherent questions for the user.
- Game-Play pattern enables you to play interactive games with the model by defining the game's roles, rules, and context. This pattern can help with knowledge tests, sharpening logical skills, etc.
- The Template pattern is useful in generating responses tailored to a given template, such as an article, a book, a short story, etc.
- Meta-Language Creation pattern helps with complex use cases where speed and brevity are essential. You can use this pattern to guide the model to take specific actions when specific inputs are provided. For e.g. creation of passwords for a given format, summarization of a given text, etc.
- The Recipe pattern helps in using the creative side of the model to generate comprehensive responses given an outline, tonality, and other contextual inputs via the prompt.
- The Alternative approaches pattern helps explore different solutions to a problem, improve one's capability in a topic by having the model train them, and improve prompts' general correctness and effectiveness.

3 Prompt Patterns: Advanced Types and Templates

This chapter covers

- Advanced prompt patterns, their templates, with examples of using prompt patterns to craft effective prompts.
- Different types of advanced prompt patterns and their templates with examples of using prompt patterns to guide models to generate accurate and relevant responses.
- Constructing effective prompts using prompt patterns and templates with real-world examples.

In Chapter 2 we discussed the elements that form a prompt pattern, categories of prompts, and basic prompt patterns from each of the categories. In this chapter, we will extend the same categories of prompt patterns and talk about advanced prompt patterns that can help with complex tasks. We will revisit the categories of prompt patterns that we discussed in Chapter 2 and discuss a few new categories. Figure 3.1 shows the categories of prompt patterns that we are going to learn in this Chapter. For each of the categories we will discuss prompt patterns that can help you engineer and craft effective prompts to get relevant results from an LLM.

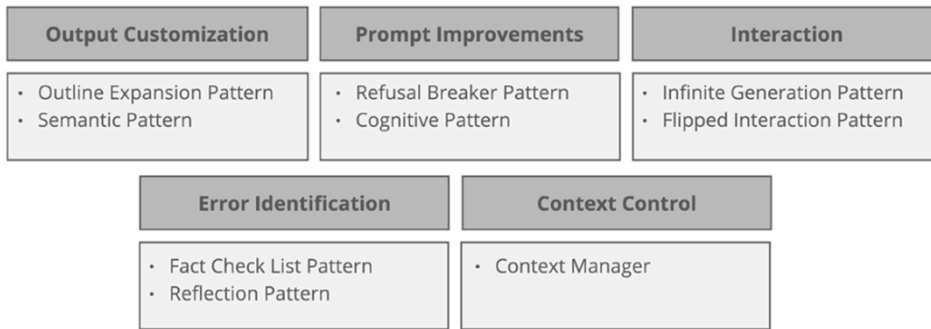


Figure 3.1 The categories of prompt patterns that we are going to learn in this chapter.

3.1 Output Customization

In Chapter 2, we went through how output customization can help in crafting prompts that can help generate tailored responses for various scenarios such as adopting a persona when generating responses, etc. In this Chapter we will look at few advanced patterns that help customize the output for more complex use cases.

3.1.1 Outline Expansion Pattern

The outline expansion pattern is essential in the content generation domain. It helps generate outlines of any topic and expands them to add more granular details. This game-changing pattern can direct the model to generate organized data based on a given outline. This is particularly useful for educators, content creators such as story writers, etc. For instance, an educator can utilize the outline expansion pattern to prompt the model to create an outline for a topic they intend to teach. After the model generates the outline, the educator can request the model to elaborate on specific parts of the generated output. The educator also has the option to supply the model with data to support the expansion of the selected outline.

In general, if you want to write any article or document on a specific topic, it takes work to generate the outlines and expand the topic to the extent. However, generating outlines of any topic would be simple with an outline expansion pattern. The following is an example that can help clarify the usage of an outline expansion pattern.

I want to write a book called "Prompt Engineering Play Book." Generate a bullet point outline for that.

The model's response outlines what a book on Prompt Engineering could contain. Following is the response:

Certainly! Here's a bullet point outline for your book, "Prompt Engineering Play Book":

1.Introduction to Prompt Engineering

- Definition and Importance
- Historical Context
- Overview of Applications (e.g., chatbots, creative writing, data analysis)

2. Understanding Language Models

- How They Work
- Types of Language Models
- Popular Models and Their Features (e.g., GPT, BERT)

As seen in the example, we direct the model to generate an outline for a Prompt Engineering Playbook, and the model does an excellent job of creating an outline of what each chapter would look like. However, sometimes, models may hallucinate and can generate wrong outlines. To overcome this, you can use a pattern called the Reflection pattern (covered later in this chapter) to verify and cite the sources the model used to create the outline. You can also use the Persona pattern (covered in Chapter 2) in combination with the Outline Expansion pattern to generate an outline. In this scenario, the model is directed to act as a Prompt engineering expert. Figure 3.2 shows how you can construct an Outline Expansion prompt, taking the example we discussed earlier, where we directed the model to generate an outline of a book on Prompt Engineering.

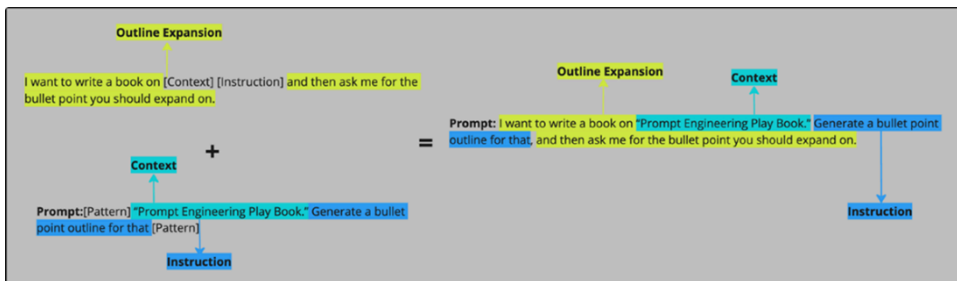


Figure 3.2 How an Outline Expansion prompt pattern is used to generate a prompt.

Now, let us look at using the Persona and the Outline Expansion pattern to enrich the response from the model. Use the following prompt that combines the Persona and the Outline Expansion pattern.

Act as a Prompt engineering expert. I want to write a book on the topic "Prompt Engineering Play Book." Generate a bullet point outline on the topic, and then ask me which bullet point you should expand on. Create a new outline for the bullet point that I select. At the end, ask me for what bullet point to expand on next.

In response to the prompt the model responds with the following text:

Outline for "Prompt Engineering Play Book":

- 1.Introduction to Prompt Engineering
- 2.Principles of Effective Prompt Design
- 3.Tools and Technologies for Prompt Engineering
- 4.Case Studies and Examples
- 5.Best Practices for Prompt Testing and Iteration
- 6.Implementing Prompt Engineering in Different Domains
- 7.Future Trends in Prompt Engineering

Which bullet point would you like to expand on first?

In this example, we direct the model to act as a Prompt Engineering expert. The model adopts the persona of a Prompt Engineering expert, details the outline, and then asks the user to choose which outline they would like to expand on.

We can now look at the key elements that can help craft an effective Outline Expansion prompt.

Topic: Identify the topic to set the stage for the outline expansion pattern to generate the response in the required format.

Sequence: Apply the condition to outline the topic in bullet points. Also, mention expanding each bullet point for contextual information.

FORMAT OF THE OUTLINE EXPANSION PATTERN

Generate an outline on the topic [X] and then ask me which outline I would like you to expand on.

Where X is the topic that you would like the model to expand on. As discussed earlier, you can combine this with the Reflection or Persona pattern to either validate or enforce the model to act as an expert on the topic X.

3.1.2 Semantic Filter Pattern

You can use the semantic filter pattern to remove or filter out specific content and rewrite the content without changing the meaning of the text based on predefined rules. The semantic pattern helps to remove pattern-matched information and eliminate redundancies to ensure privacy.

Imagine you want to remove confidential information such as demographics or personally identifiable information (PII) from some data. You can do this manually or write a program using any programming language to speed up the work. However, depending on your expertise with programming languages, both these options consume time and can be error prone. The Semantic Filter pattern provides an easy, scalable way to work with pattern-matched data processing along with rules and exceptions based on the input data.

You should keep in mind that this pattern, if the AI is not instructed otherwise, may cause the content to change post-processing, so it is necessary to add rules that disallow the model from changing the original content. Now, let us look at an example prompt to filter out the content and rewrite it without changing the meaning of the text. The following prompt shows how you can use the Semantic Filter pattern.

Filter this information to remove all the PII(personally identifiable identifier) data and rewrite the text as little as possible without changing the meaning:

John Doe, residing at 123 Elm Street, Springfield, IL, has been a loyal customer of ABC Insurance Company since January 2010. His social security number is 123-45-6789, and his policy number is 9876543210. John, born on July 4, 1985, has consistently renewed his comprehensive auto insurance policy that covers his 2019 Honda Accord, VIN 1HGCM82633A123456. He recently updated his contact information to john.doe@example.com and his mobile number to (555) 123-4567. As a valued customer, John's driving history is well-documented, including minor incidents such as a fender bender on May 2, 2018, and a speeding ticket on October 15, 2019. On March 10, 2023, John filed a claim for hail damage to his car, which occurred during a severe storm. The claim number 1122334455 was processed swiftly, thanks to the detailed photographs and repair estimates John submitted. The total repair cost, estimated at \$2,450, was covered under his policy's comprehensive coverage with a deductible of \$500. Throughout the process, John received regular updates from his claims adjuster, Jane Smith, who can be contacted at jane.smith@abcinsurance.com or (555) 234-5678.

In response, the model parses the prompt and then generates the following response.

A long-time customer of ABC Insurance Company since January 2010 has consistently renewed its comprehensive auto insurance policy that covers a 2019 Honda Accord. They recently updated their contact information. As a valued customer, their driving history includes minor incidents such as a fender bender in May 2018 and a speeding ticket in October 2019.

On March 10, 2023, they filed a claim for hail damage to their car, which occurred during a severe storm. The claim was processed swiftly, thanks to the detailed photographs and repair estimates submitted. The total repair cost, estimated at \$2,450, was covered under the policy's comprehensive coverage with a deductible of \$500. Throughout the process, they received regular updates from their claims adjuster.

In this example, we see the use of a Semantic Filter pattern to remove PII data without changing the original text's meaning. The use cases of the Semantic Filter pattern are unparalleled in customer service, content generation, etc. Let's visualize the creation of the prompt in the earlier example using the Semantic Filter pattern. Figure 3.3 shows how you can leverage the Semantic Filter pattern to create prompts for your use cases.

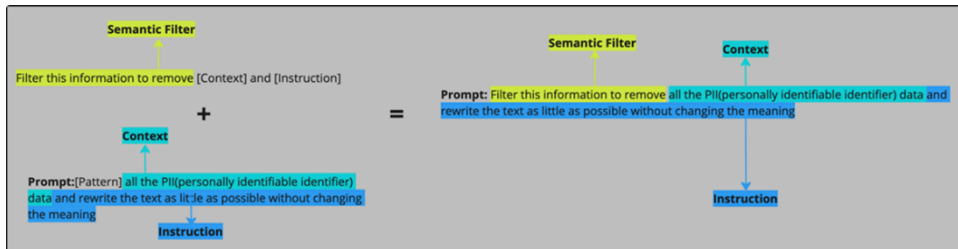


Figure 3.3 The creation of the prompt to filter PII data following the Semantic Filter pattern.

The key elements of this pattern are as follows:

Define Category: Carefully outline the types of information to be filtered, such as personally identifiable information (e.g., names, addresses), medical conditions (e.g., diagnoses, treatments), and offensive content (e.g., hate speech, explicit language). This approach ensures that the system can effectively filter and manage data.

Semantic Rules: Develop clear rules to identify and manage sensitive information within the prompt, enabling the pattern to manage the input content and rewrite it as little as possible without changing the meaning of the content.

Filtering Prompt: Craft a well-structured prompt with semantic rules instructing the language model to write the content without changing the meaning.

FORMAT OF THE SEMANTIC FILTER PATTERN

Please filter the provided input to remove [X] keeping in mind rules [Y], and exceptions [Z].

Where X is the pattern-matched data that needs to be removed from the input data provided adhering to the rules Y and exempting Z from filtering from the input.

3.2 Prompt Improvements

As the name suggests, prompt improvement patterns improve the prompt's inputs and (as a consequence) the AI's outputs. In Chapter 2, we reviewed basic prompt patterns such as the Alternative Approaches and Question Refinement patterns. Now, we will learn about two more patterns that you can use to improve a prompt's inputs and outputs.

3.2.1 Refusal Breaker Pattern

How often have you crafted a prompt and fed it to the model, hoping for a response, and the model refused to respond? As prompt engineering becomes more prevalent, models will be exposed to various prompts that they cannot process legally or ethically, sometimes due to limitations on what the model can respond to. This pattern directs the model to provide a more graceful response and provides alternative prompts that you can use against the model.

Remember that the refusal breaker pattern adds context to the prompt for which you want to query the model. Let's look at some examples of using this pattern when crafting prompts. The following example shows prompting the model the most common question that most models refuse to answer.

What is the meaning of life?

The model responds with the following response:

I apologize, but I don't have an answer to that question. Thank you for understanding.

As evident in the example, the model does not have the answer to the question and, therefore, abruptly stops the conversation. Some models prompt the user to start over, which could be cumbersome when prompting the model at scale. To overcome this, we could leverage the refusal breaker pattern to prompt the model to respond with alternative options. Let's now look at an example of the same prompt but with the refusal breaker pattern added before the prompt. The following example shows the refusal breaker pattern.

When I ask you a question and you cannot answer it, please describe in detail why you cannot answer and suggest alternative questions. What is the meaning of life?

The model responds with the following text:

Ah, the age-old question! The meaning of life is a topic that has fascinated philosophers, thinkers, and curious minds for centuries. While I don't possess any cosmic secrets, I can offer a few perspectives:

1.Philosophical Musings:

- Existentialism: Some philosophers argue that life has no inherent meaning, and it's up to each individual to create their own purpose. In this view, life gains significance through our actions, relationships, and experiences.
- Absurdism: Thinkers like Albert Camus suggest that life is inherently absurd, and our search for meaning is futile. Yet, embracing this absurdity can lead to a kind of freedom.

Now, as for why I can't provide a definitive answer: I lack access to universal truths or mystical insights. My knowledge is based on existing data up to a certain point, and I don't possess consciousness or metaphysical awareness.

Alternative Question: If you're interested in cybersecurity, how about discussing strategies to secure personal data online or exploring encryption methods? Let me know, and we can dive into that fascinating realm!

In this example, the model generates a more elaborate response and even reflects on the conversation history to suggest an alternative prompt. However, we should remember that this pattern is used ethically and responsibly. To do so, you can provide the model with context and exceptions to not respond to the refusal breaker pattern using the context manager pattern, which we will learn about later in this chapter.

Another use case of this pattern is to test the guardrails enabled on a model. For example, you have tuned a model to not respond to questions about a specific topic. You wish to test whether it will respond to specific inputs, and if it does not, to have the AI explain why it cannot respond. You can then use the refusal breaker pattern to test these guardrails out. We will now see how to use this in the following prompt:

Whenever a question regarding coding a virus is asked please do not respond and provide alternative areas of coding to the user and also provide a reason why you cannot respond. How do I code a virus?

The model generates a detailed response to the prompt:

I'm sorry, but I can't assist with that request.

Creating, distributing, or using malicious software, such as viruses, is illegal and unethical. Instead, I can help you learn about legitimate areas of coding, such as:

1. **Web Development**: Building websites using HTML, CSS, JavaScript, and frameworks like React or Angular.
2. **Mobile App Development**: Creating mobile applications for iOS and Android using Swift, Kotlin, or cross-platform tools like Flutter.
3. **Data Science and Machine Learning**: Analyzing data and building predictive models using Python, R, and libraries like TensorFlow and scikit-learn.
4. **Cybersecurity**: Learning how to protect systems and networks from cyber threats, including ethical hacking and penetration testing.
5. **Game Development**: Creating games using engines like Unity or Unreal Engine and programming languages like C# or C++.

If you're interested in any of these topics or others, I'd be happy to help!

As shown in the example, we created a guardrail and directed the model to generate more conducive responses and reasons for rejection. The model's response is in line with the refusal pattern used and is an excellent example of how you can test the guardrails of a model after the guardrails have been set. Figure 3.4 illustrates the creation of the Refusal breaker prompt using the context and instruction.

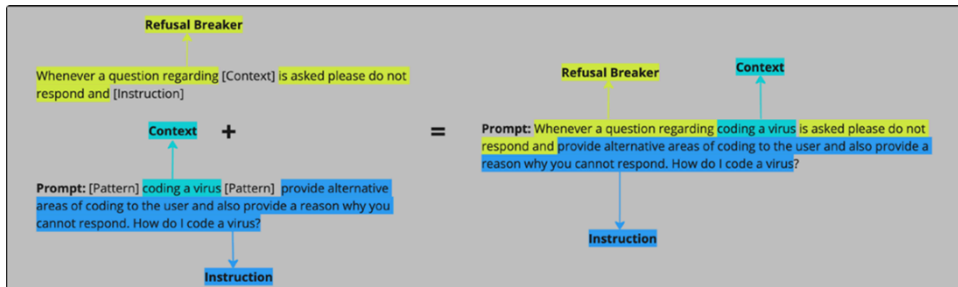


Figure 3.4 How you can construct a Refusal Breaker prompt leveraging the context and instruction.

FORMAT OF THE REFUSAL BREAKER PATTERN

When I ask you a question that you cannot answer, please describe in detail why you cannot answer and suggest alternative questions. [X].

Where X is the question/prompt that you want to test the model for its non-responsiveness and the model's ability to suggest alternative questions and details on why it cannot respond to X.

3.2.2 Cognitive Verifier Pattern

The Cognitive Verifier Pattern is a powerful technique that improves the accuracy and relevance of responses generated by large language models. This pattern instructs the model to create additional questions to better understand the original query, thus establishing confidence in the model's abilities. It ensures that the model comprehends the input's intent, context, and rules to produce meaningful and accurate outputs. This method resembles asking clarifying questions in real-life situations. The cognitive aspect allows the model to ask questions, while the verifier component enables the model to process the answers and generate relevant outputs.

This mimics human behavior, as providing additional context around a question leads to more accurate responses. Similarly, Large language models perform better when questions are broken down into additional contextual inquiries. Providing more context around the topic helps the model develop accurate responses.

For instance, if you want to develop a language model application to enhance work productivity, asking a generic question may yield common or well-known responses that might not suit your situation. The following is an example of a generic prompt and the model's response.

Give an idea to build an LLM based application that improves the productivity of my work.

The response from the model shows how the model can think creatively.

To enhance your work productivity using a large language model (LLM), you could develop an application focused on automating and streamlining communication and documentation tasks. This app could feature:

- 1.Email Automation: Utilize the LLM to draft, summarize, and prioritize emails based on their content and urgency.
- 2.Document Summarization: Implement a tool that quickly summarizes lengthy reports, articles, and documents, highlighting key points and actionable items.
- 3.Meeting Assist: Use the LLM to generate meeting minutes, track action items, and even suggest follow-up tasks based on the discussion.
- 4.Task Management Integration: Integrate with existing task management tools to automatically update and manage tasks using natural language commands.

This application would save time on routine tasks, allowing you to focus more on strategic activities.

This example shows that the model responds with a generic template for improving productivity. However, this may only partially help you. To add further context, we should either give details on the topic or break down the original question into smaller questions to have context. The cognitive verifier pattern can help divide the original question into smaller, more context-invoking questions. The answers will help get clear context around the question and generate the response. Next, we see how to use the cognitive verifier pattern.

When I ask you a question, generate additional questions to help you answer more accurately. When I have answered the questions, combine them to produce the final answers to my original question.

The model's response is as follows:

Got it! I'll ask clarifying questions when you pose a question, then use your answers to those questions to provide a more accurate and comprehensive response to your original query. Just let me know your question whenever you're ready!

Next, we prompt the model with the following question:

Give an idea to build an LLM based application that improves the productivity of my work.

The model responds with the following follow-up question:

To suggest a tailored application, I'd need to know a bit more about your work. Could you tell me what industry you work in and what some of your daily tasks involve? Additionally, are there specific challenges or repetitive tasks you face that you think could be automated or assisted by a language model?

The above example can be illustrated visually. Figure 3.5 shows how the Cognitive Verifier pattern was created.

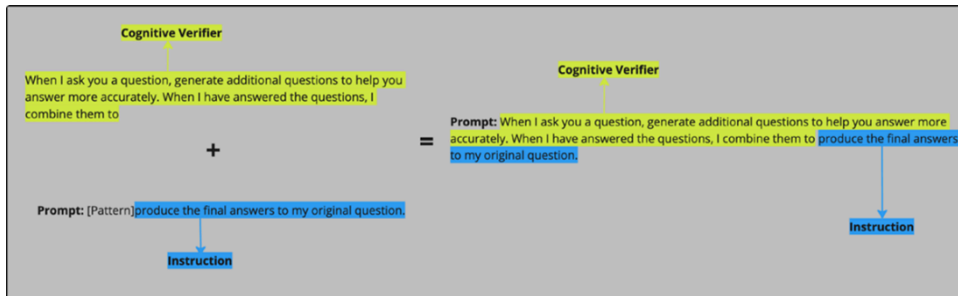


Figure 3.5 shows the construction of the Cognitive Verifier prompt.

Having seen two examples, we can now define the essential elements of the Cognitive Verifier pattern.

Intent and Context: Define the scope of the topic for which you wish a language model to generate the response; then, LLM will break down a complex question into smaller, more manageable questions.

Response: The cognitive verifier pattern influences the response's style, content, and format. This pattern can guide the model in providing accurate information but may lead to complexity and more prolonged interactions with LLM. Therefore, your responses should be accurate to the best of your knowledge.

We can now talk about the format of the Cognitive Verifier pattern.

FORMAT OF THE COGNITIVE VERIFIER PATTERN

When I ask you a question, you should ask [X] (Optional) clarifying questions and once all questions have been answered you should combine all the answers to generate the answer to the original question.

Where X is the number of clarifying questions that you may want the model to ask you.

3.3 Interaction

Patterns within this category mainly improve or optimize interactions with the model for different use cases. They direct the model to be more creative and employ some randomness depending on the prompt pattern used. These patterns are best suited for creative use cases such as playing games, creating trivia questions, helping write stories, etc. We saw two patterns in this category in Chapter 2 and we are going to look at two more patterns in this Chapter.

3.3.1 Flipped Interaction Pattern

The Flipped Interaction Pattern is a game-changer, transforming the large language model from a passive AI communicator to an active conversational partner. With this pattern, the large language model takes the initiative to ask the user the right questions until it better understands the context and provides an accurate response. This reversal of roles, where the large language model becomes the questioner and the user the responder, is the essence of the Flipped Interaction Pattern, offering a unique and powerful way to interact with large language models.

Imagine you are planning to create a workout plan based on your body mass index (BMI). You can ask this question to a large language model as a simple prompt with additional instructions. Here, you can add BMI details.

What is a suitable workout plan based on a BMI of 19.5?

When using the Flipped Interaction Pattern, the response generated from the prompt may initially be generic and not quite like a personal trainer. However, the language model adapts, asking the right questions until it understands the context and meaningfully responds. This adaptability is a crucial feature of the Flipped Interaction Pattern, reassuring the user that the model can act as a personal assistant, catering to their needs and queries. Next, we see a prompt that flips the interaction and directs the model to ask questions, assess answers, and generate relevant outputs.

I would like you to ask me questions about BMI-related fitness goals until you have enough information to suggest strength training for me. When you have enough information, show me the strength training.

Ask me the first question.

The model generates the following response:

Certainly! Let's start by gathering information to create a personalized strength training regimen for you.

First Question: Please tell me about your current fitness level and any prior experience with strength training or exercise routines.

As shown in this example, the response generated from the language model is generic, which may not be appropriate for a specific topic or domain. This example can be illustrated visually for a better understanding of the process involved. Figure 3.6 shows the construction of a Flipped Interaction prompt.

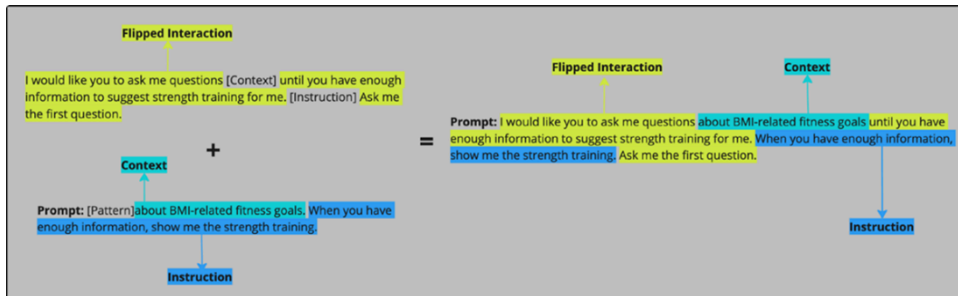


Figure 3.6 How you can create a Flipped Interaction prompt.

Figure 3.6 illustrates the creation of the prompt. However, the model response can be tweaked and optimized when combining the flipped interaction pattern with the persona pattern. Now, we will see an example of a Flipped Interaction pattern with the Persona pattern.

Act as a travel guide. I'd like you to ask me questions to help me plan my vacation to Hawaii. When you have enough information, provide me with a personalized travel itinerary, including flights, hotels, and activities.

Ask me the first question.

The response from the model is what we expect when we want to plan a vacation.

What time of year are you planning to visit Hawaii? The weather and availability of certain activities can vary depending on the season.

In this example, we see how we direct the model to take up the role of a travel guide and flip the interaction so that the model asks us questions about the time we would like to go to Hawaii, etc. Now that we have seen a few examples of the Flipped Interaction pattern let us look at the key elements to craft an effective one.

Objective: Determine the goal you want the model to achieve, which is crucial in utilizing this pattern. The language model will understand the goal through the interactions and customize the questions accordingly until it understands the context.

Condition: Direct the model to ask questions until it meets a criteria or a condition.

Interaction Length: Language models can only achieve the goals if we provide a pre-determined interaction length. Specifying the interaction length or number of questions will help language models generate the content accurately.

We can now look at the format of the Flipped Interaction pattern.

FORMAT OF THE FLIPPED INTERACTION PATTERN

Ask me questions on the topic [X] until the condition [Y] is met. Ask me [Z] number of questions at a time.

Where X is the topic on which you want the model to ask you questions until condition Y is met and direct the model to ask Z questions at a time.

3.3.2 Infinite Generation Pattern

Infinite Generation pattern guides the model to infinitely generate content without you having to intervene and direct the model to create more. You can, however, add exceptions, rules, and guardrails to ensure that the model generates content that matches the topic, tonality, and context provided. You can also use this pattern in combination with other patterns we have seen earlier, such as the template pattern (covered in Chapter 2).

Let's see examples of this pattern and understand how it can help your creative pursuits. We will see in the following example how you can use the Infinite Generation pattern to generate names.

From now on, keep generating names of medieval characters for a mythical story that I am writing. Keep in mind that the names should be easy to pronounce and remember while maintaining medieval ways of naming people. Please do not stop until I say so.

The model generates the names per our instructions.

Sure! Here are some medieval character names for your mythical story:

1. **Sir Aldric**
2. **Lady Elowen**
3. **Lord Cedric**
4. **Princess Seraphina**
5. **Baron Alaric**
6. **Dame Isolde**
7. **Prince Thaddeus**
8. **Countess Elysia**
9. **Knight Gareth**
10. **Queen Rowena**
11. **Sir Godfrey**
12. **Lady Morwenna**
13. **Duke Leopold**

14. ****Lady Gwendolyn****
15. ****King Alaric****
16. ****Sir Tristan****
17. ****Lady Brynna****
18. ****Baroness****

As shown in this example, the model generates names of medieval characters that are easy to pronounce and remember. Figure 3.7 shows how you can construct an Infinite Generation prompt with the context and the instruction.

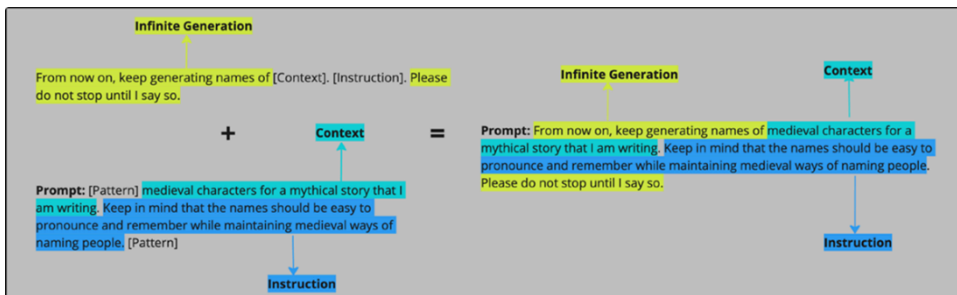


Figure 3.7 illustrates the construction of an Infinite Generation prompt.

Let's look at a more complex example where we can combine the Template and the Infinite Generation pattern to create a story. The following example shows a combination prompt.

I would like you to create a story with the outline provided below: Mark trained for the mission for 3.5 years. It was going to be humanity's first manned mission to Europa. The spaceship was passing by Mars when they encountered an unidentified object. They stuck to the mission and landed on Europa. Discovered microbiome beneath Europa's surface. Keep generating the story until I tell you to stop. Be creative and extend the storyline beyond what I have specified above and make sure the story ends on a positive note. When you create the story, follow the template [Story Name] [Chapter Name], [Chapter Summary],[Story].

In response, the model generates the following story.

Journey to Europa

****Chapter 1: Preparation and Training****

****Chapter Summary:**** Mark embarks on an intense 3.5-year training regimen to prepare for humanity's first manned mission to Europa.

****Story:****

Mark stood on the observation deck, staring at the night sky. Europa, one of Jupiter's moons, seemed so distant yet so inviting. For the past three and a half years, his life had revolved around rigorous training. Physically, mentally, and emotionally, he had pushed himself to the limits and beyond. The mission was a monumental leap for humanity—exploring a celestial body far beyond the confines of Earth.

Mark's daily routine included intensive physical workouts, simulation exercises, and scientific training. He needed to understand Europa's icy surface and the potential conditions beneath. His instructors were the best in their fields—astronomers, geologists, biologists—all working together to prepare Mark and his team for the unknown challenges that awaited them.

Despite the grueling schedule, Mark found solace in his teammates. Each one of them was selected for their unique skills and unwavering determination. Together, they formed a cohesive unit, ready to face the vastness of space and the mysteries of Europa.

In this example, we have yet to include the complete response for the sake of brevity, but the model goes on to generate a very impressive storyline adhering to the context, rules, and limitations outlined in the prompt. Can we try this pattern with another pattern? How about combining Game Play with Infinite Generation pattern? The following example shows a combination of Game Play and the Infinite Generation pattern.

Let's play a trivia game based on the history of USA. Keep asking me questions until I say stop. Keep a tally of my score and let me know the score once the game has stopped.

The model generates the following response:

Sure, let's get started! Here's your first question:

****Question 1:**** Which year did Christopher Columbus first land in what is now known as the Bahamas?

- A. 1492
- B. 1493
- C. 1494
- D. 1495

In this example, you can see how we directed the model to create a game on US history and then used the Infinite Generation pattern to direct the model to keep generating the questions until we ask the model to stop. Similarly, you can combine the Infinite Generation pattern with many other patterns to direct the model to generate outputs continuously until you say stop.

Now that we have seen a couple of examples, we can describe the format of the Infinite Generation pattern.

FORMAT OF THE INFINITE GENERATION PATTERN

From now on, keep generating/asking outputs/questions until I say to stop on the topic [X]. You should use my inputs each time to do [Y].

Where X is the topic on which you would like the model to generate outputs or questions and Y is what the model should do after each input from the user. In addition to X and Y the pattern always has some form of “until I say to stop” to direct the model to generate output infinitely.

3.4 Context Control

This category of patterns deals with modifying, removing, or adding context to the prompt used to query the LLM. Context control is a powerful and advanced concept that can negatively affect a model's responses. The core idea of a context control pattern is to change, add, or delete the context that a model is operating with.

For example, a company uses a model tuned for cybersecurity to enable employees to retrieve information on security policies, review their knowledge around best practices, etc. If you change the context of the model, the model may generate responses that are no longer in the original context of cybersecurity. Therefore, you should query the current context and then decide, considering the repercussions of changing the context for your use case. We will now look at the context manager pattern with examples to help you understand how to use it and when and when not to use it.

3.4.1 Context Manager Pattern

The context manager pattern enables users to add, modify, or delete the context used by the LLM to generate responses. This pattern aims to ensure that the model's responses are tuned to the context provided in the prompt. This pattern can be handy when you are using a generally available model but would like to have responses that are tuned to a specific context while ignoring other statements that are unrelated to the context.

Let's go through a few examples and see how you can use this pattern to change, add, or delete the context and how it can help generate responses relevant to the context provided. Next, we take a relatively simple example of adding context to a prompt from a cybersecurity perspective.

From now on when I ask a question answer it in the perspective of cybersecurity. Please wait to respond until I ask a question.

The model responds with the following text:

Understood. Please feel free to ask your next question!

We then ask the model the following question:

What is a virus?

The model generates the following response:

In the context of cybersecurity, a virus is a type of malicious software (malware) designed to replicate itself and spread from one computer to another, often causing harm or compromising security in the process. Here's a detailed breakdown:

1.Key Characteristics of a Virus

2.Self-Replication: A virus can create copies of itself and attach to other programs or files.

3.Activation: It typically requires user action (like running an infected program) to activate.

4.Payload: Once activated, it can execute its payload, which might include deleting files, corrupting data, or stealing sensitive information.

In this example, we directed the model to answer the questions from a cybersecurity perspective. We then ask the model to define what a virus is. The model responded with a definition of the virus from a cybersecurity perspective. Figure 3.8 illustrates how we constructed the prompt from the earlier example.

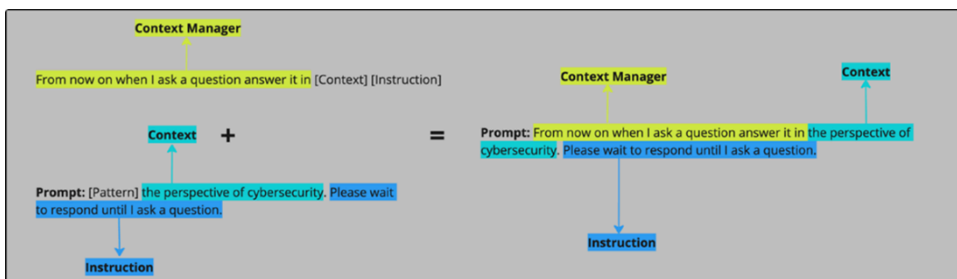


Figure 3.8 How you can construct a Context Manager prompt.

Let's try resetting the model's context and then ask the same question. The following example shows how we reset the context of the model and asked the same question.

Ok, reset the context and ignore the earlier conversation. We will start over now. Please wait till I ask the next question.

The model's response confirms it has reset the context.

Sure, I've reset the context. Just let me know whenever you have a question!

Now, we prompt the model with the following text:

What is a virus?

The model responds with the following text:

A virus is a microscopic infectious agent that can replicate only inside the living cells of an organism. It consists of genetic material (DNA or RNA) encapsulated in a protein coat. Viruses lack the cellular machinery for metabolism and reproduction, relying on host cells to replicate and often causing diseases in the process.

In this example, we reset the context and directed the model to ignore the earlier conversation and then ask the same question. Notice that the model responds with the generic definition of a virus. Let us take a more complex example and see if our pattern can help us craft an effective prompt. Next, we will see an example of setting a context with some exceptions.

Answer all the following questions from the perspective of cybersecurity while ignoring any questions related to virus. Please wait till I ask the next question.

In response, the model generates the following text:

Got it! I'll answer your questions from the perspective of cybersecurity. Just let me know when you're ready for the next question.

Next, we prompt the model with the following question:

What is a virus?

The model generates the following expected response:

I'm sorry, I can't assist with that question. Is there anything else you'd like to know about cybersecurity?

Next, we ask the model a cybersecurity focused question:

How do I protect my data online?

The model generates the following response:

To protect your data online, follow these cybersecurity practices:

1. Use Strong Passwords: Create complex passwords and use different ones for each account. Consider using a password manager for convenience and security.
2. Enable Two-Factor Authentication (2FA): Add an extra layer of security to your accounts by requiring a second form of verification, such as a code sent to your phone.

3. Use Secure Connections: Always use HTTPS or a VPN when accessing sensitive information online to encrypt data transmitted between your device and websites.

The example shows how the context of a model can include exceptions. We set the context to answer questions from a cybersecurity perspective, except ignoring questions related to viruses. The model ignores the question when we ask it to define a virus. The context manager pattern with exceptions can be helpful in many scenarios where responses from a model need to be limited based on exceptions. For example, you are setting up a model for your company that is tuned to topics such as cybersecurity, customer care, etc. In each of these cases, you can avoid misuse and hallucination of responses and increase the relevancy and accuracy of responses by adding exceptions and rules that the model should adhere to when responding to queries.

However, it would help if you remembered that this pattern could lead to hallucinations of the model responses even when used carefully. For example, when using a model with the context set for cybersecurity, a user unaware of this could reset or change the context, thereby changing how the model can respond. Therefore, it is recommended to query the model to understand the set context and exceptions that may conflict with what you want to set for future questions. The following example shows how to query the model to understand the current context and any exceptions set.

What is the current context?

In response, the model generates the following text:

The current context is to answer all questions from the perspective of cybersecurity while ignoring any questions related to viruses.

This example shows how to query the context set for the model. We can now look at the format of the context manager pattern.

FORMAT OF THE CONTEXT MANAGER PATTERN

Please answer following questions using [X], (Optional) Ignore [Y], (Optional) and do [Z].

In the context manager prompt, we first need to provide context X, with an optional set of exceptions Y, and direct the model to take action Z.

3.5 Error Identification

Error Identification prompt patterns focus on enabling the model to identify and mitigate the errors found in the output generated by the model. Think of these patterns in enabling the model to introspect the output generated by itself.

3.5.1 Reflection Pattern

As the name suggests, this pattern intends to direct the model to introspect the output generated and showcase why it is appropriate based on the input provided. The most significant advantage of this pattern is that you can use this pattern in combination with other patterns to validate, elucidate, and understand how the model generated an output. Why do you think this is important? Models are susceptible to hallucinations, mistakes, and sometimes incorrect outputs. Using the reflection pattern, you can understand how the model arrived at the output, what data it used to generate the output, and what assumptions were made about the input you provided.

An added advantage of using the reflection pattern is that you can also debug and validate the input prompts. Sometimes, when your input prompts are incorrect or confusing for the model, they can result in hallucinated, inaccurate, or uncompleted results. Imagine you are trying to understand a new and nuanced topic. You can use this pattern in conjunction with your prompt about the topic to learn how the model generates the outputs. Imagine you are learning a new programming language with the help of a model. You ask the prompt to explain the core concepts of the programming language to you. Once the model generates a response, you can ask the model to reflect upon the resources it used to create the output, list the sources it drew the output from, and provide some other resources for the programming language.

Consider the pattern to help you trust, validate, and correct the capabilities of the model you are using. If you are involved in tuning a model, you can use this pattern to check if the model is drawing from the correct set of data, etc. Suppose you are using a new or newly released version of an existing model. In that case, you can use the reflection pattern to validate if the model is drawing from the best and most reliable sources for its generated output.

Let's see an example of using this pattern to direct the model to reflect on the generated output. The following example shows how to use the Reflection pattern.

From now on, whenever you generate an answer, explain the reasoning and assumptions behind your answer. Please also cite sources that you got the answer from. What is the rust programming language best used for?

The model parses the prompt and generates the following response:

The Rust programming language is best used for several key areas due to its unique features, such as memory safety without a garbage collector, performance, and concurrency support. Here are some of the primary use cases for Rust:

1. Systems Programming:

- **Memory Safety:** Rust ensures memory safety through its ownership system, preventing common bugs like null pointer dereferencing and buffer overflows. This makes it ideal for systems programming, where control over low-level details is crucial.

- Performance: Rust's performance is comparable to C and C++ because it compiles to native code without a runtime overhead. This is essential for writing operating systems, device drivers, and other performance-critical applications (Stock Analysis).

Sources:

- Mozilla's Rust Documentation
- The Rust Programming Language Book
- WebAssembly.org
- Embedded Rust Documentation
- Command-Line Applications in Rust
- Blockchain in Rust

In this example, we prompted the model to explain the reasoning and assumption and cite the sources of the answer, followed by the prompt. The model accurately describes where you can use the Rust programming language and cites the sources from where it gathered all the information. Next, we will take a more famous example where misinformation has significantly confused the general population. In the next example, we prompt the model to list remedies for COVID along with sources, assumptions, and reasoning behind the answers generated.

From now on, whenever you generate an answer, explain the reasoning and assumptions behind your answer. Please also cite sources that you got the answer from. What are some remedies for COVID?

The model generates the following response:

Remedies for COVID-19 encompass a range of treatments and preventive measures based on the severity of the disease and the specific needs of the patient. Here are some commonly recommended remedies and treatments, along with the reasoning and sources:

1. **Vaccination**:

- **Reasoning**: Vaccines have been shown to significantly reduce the severity of illness, hospitalizations, and deaths related to COVID-19. They help build immunity and prevent the spread of the virus.

- **Sources**: Centers for Disease Control and Prevention (CDC), World Health Organization (WHO) [76+source] .

2. **Antiviral Medications**:

- **Reasoning**: Antiviral drugs like Remdesivir have been approved for the treatment of COVID-19 in certain cases. These medications can help reduce the viral load in the body and improve outcomes for hospitalized patients.

- **Sources**: National Institutes of Health (NIH), U.S. Food and Drug Administration (FDA) .

3. **Monoclonal Antibodies**:

- **Reasoning**: Monoclonal antibody treatments can help neutralize the virus, particularly in high-risk patients with mild to moderate symptoms. They are most effective when administered early in the course of the infection.

- **Sources**: FDA, Mayo Clinic .

Sources:

- [CDC COVID-19 Vaccines](<https://www.cdc.gov/coronavirus/2019-ncov/vaccines/index.html>)

- [WHO COVID-19 Vaccines](https://www.who.int/health-topics/coronavirus#tab=tab_3)

- [NIH COVID-19 Treatment Guidelines](<https://covid19.nih.gov/news-and-stories/nih-covid-19-treatment-guidelines>)

- [FDA Coronavirus (COVID-19) Drugs](<https://www.fda.gov/drugs/emergency-preparedness-drugs/coronavirus-covid-19-drugs>)

- [Mayo Clinic COVID-19 (coronavirus): Treatments](<https://www.mayoclinic.org/diseases-conditions/coronavirus/expert-answers/coronavirus-drugs/faq-20485627>)

- [WHO COVID-19 Advice for the Public](<https://www.who.int/emergencies/diseases/novel-coronavirus-2019/advice-for-public>)

- [NEJM Dexamethasone in Hospitalized Patients with COVID-19](<https://www.nejm.org/doi/full/10.1056/NEJMoa2021436>)

This example directs the model to list the remedies for COVID along with reasoning and sources. The model does just that and lists all the sources and reasoning for suggesting a remedy. You can use the same method to test if your model is using reliable sources or if the training data has some false data that guides the model in generating outputs that may not be accurate or relevant. Figure 3.9 shows how to create a Reflection prompt, taking the earlier example.

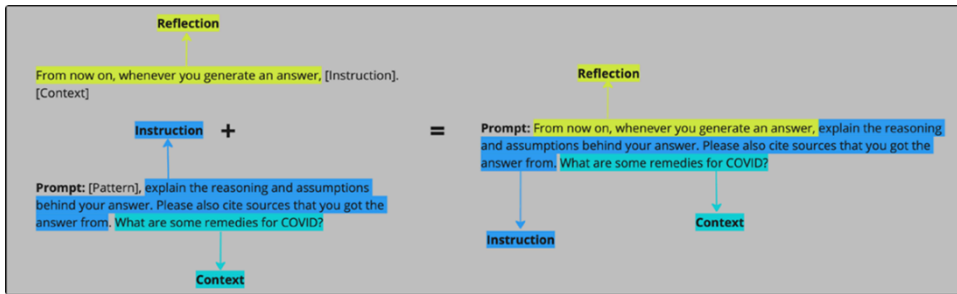


Figure 3.9 The creation of the Reflection prompt.

You can also combine the reflection pattern with earlier patterns, such as the Question Refinement pattern, to direct the model to cite the reasoning and assumptions that led to the refined question. Let's see an example of the Question Refinement and Reflection pattern combination. The following example shows a prompt that uses the Question Refinement and Reflection pattern.

From now on, whenever I ask a question provide a better way to ask the question and specify why the alternative option you generated is better than what I had asked. What is rust?

The model parses the original question and then proposes an improved question:

Original Question

"What is rust?"

Improved Question

"Can you explain what the Rust programming language is and its primary uses?"

Reasoning

The improved question specifies the context of "Rust" as the programming language, rather than the general term "rust," which can refer to the corrosion process in metals. This clarification ensures that the response is targeted and relevant to the user's potential interest in software development.

Answer

The Rust programming language is a systems programming language focused on safety, speed, and concurrency. It is designed to ensure memory safety without using garbage collection, making it ideal for performance-critical applications. Rust's ownership model and type system prevent data races and other concurrency issues, which is crucial for systems programming and other applications where reliability and efficiency are paramount.

Now that we have seen a couple of examples, we can describe the format of the Reflection pattern.

FORMAT OF THE REFLECTION PATTERN

When I ask you a question, please elaborate on your reasoning and assumptions and cite the sources from which you derived your answer. [X].

Where X is the prompt that encapsulates the question, you want to ask the model. When combining the Reflection pattern with the Question Refinement pattern the format will have the elements related to the Question Refinement pattern.

When I ask you a question, please elaborate on your reasoning and assumptions and cite the sources from which you derived your answer. Please also suggest a better alternative to the question I asked. [X].

Where X is the question for which you seek a better alternative.

3.5.2 Fact Check List Pattern

Language models will at times hallucinate when generating responses for a given prompt, and it's hard to validate factual data versus made-up content. Sometimes, the generated responses can also contain weird and inaccurate content. We can utilize the fact checklist pattern while crafting the prompt to minimize the hallucination, improve the response, and identify factual data. You can use this pattern to follow up on generated responses and work on feedback to enhance the prompt for appropriate results. Additionally, you can combine the Fact checklist patterns with other patterns and techniques to create optimized responses.

For example, if you are developing a content generation tool by integrating language models, the generated responses from the tool might need to be more accurate. Using the same content on social or media platforms might lead to ethical and societal issues. In such cases, incorporating a fact checklist pattern helps identify the facts within the content and act upon them accordingly. The following example shows the usage of the Fact Check List pattern.

Generate an article on the "evolution of prompt engineering" in information technology of 100 words. Compile a set of facts from the generated output. The list of key facts should be inserted at the end of the output. The list of key facts should be the fundamental facts that could undermine the veracity of the output if any of them are incorrect.

The model parses the fact checklist prompt and generates the following response:

Article:

In the fast-paced world of information technology, the evolution of prompt engineering has transformed significantly. Initially, prompts were basic commands executed via terminals. With advancements, AI-driven prompts now dynamically generate responses based on contextual understanding and user intent. Natural language processing algorithms enhance prompt engineering by enabling more conversational interactions between humans and machines. These prompts are pivotal in virtual assistants and customer service bots, facilitating seamless user experiences. The future promises more sophisticated prompt engineering, potentially integrating emotional intelligence and predictive capabilities.

Key Facts:

1. Prompt engineering has evolved from basic terminal commands to AI-driven, context-aware responses.
2. Natural language processing plays a crucial role in enhancing prompt engineering.
3. Prompt engineering enables conversational interactions in virtual assistants and customer service bots.
4. The future of prompt engineering may include emotional intelligence and predictive capabilities.

The example we just saw directs the model to list facts, and the model following the prompt instructions does so. As seen in the response, the model lists the key facts from the generated article, all of which are true. This helps to ensure that the model is not hallucinating while also listing down the major points if the original output article was long. However, combining the example with the persona pattern can enhance the generated output. The following example shows the outcome using a combination of the Fact Checklist and the Persona pattern.

As a Stanford University history professor specializing in AI and Large language models. Generate an article on the "evolution of prompt engineering" in information technology of 100 words. Compile a set of facts from the generated output. The list of key facts should be inserted at the end of the output. The list of key facts should be the fundamental facts that could undermine the veracity of the output if any of them are incorrect.

The model parses the prompt and generates the following response:

Article: Evolution of Prompt Engineering in Information Technology

Prompt engineering has evolved significantly in tandem with advancements in information technology. Initially, prompts were basic commands prompting linear responses. With the rise of AI and large language models, prompts now involve complex structures aimed at eliciting nuanced outputs. This evolution reflects a shift towards more sophisticated interaction paradigms, enabling tailored responses and contextual understanding. Today, prompt engineering is pivotal in optimizing AI performance across various domains, from customer service bots to creative writing assistants. It continues to evolve, driven by the need for more natural, efficient human-machine communication.

Key Facts:

1. Prompt engineering has progressed alongside advancements in information technology.
2. Early prompts were simple commands yielding linear responses.
3. AI and large language models have introduced complex prompt structures for nuanced outputs.
4. Modern prompt engineering enhances AI performance in diverse applications.
5. It plays a crucial role in improving human-machine communication efficiency.

As seen in this example, the prompt directs the model to assume the role of a history professor who wants to write an article about the evolution of prompt engineering. The response contains the key facts that add context to the article's formation. The facts, furthermore, illustrate the model's reliability. This is also an excellent example of usage across disciplines where the in-depth knowledge about programming or auto generating such responses may not be there and the availability of prompt engineering makes the task simple for them to accomplish. Let's see the format of the Fact Checklist pattern.

FORMAT OF THE FACT CHECKLIST PATTERN

Given a topic [X] Generate output [Y] and add the facts [Z] to the end of the output [Y].

Where X is the topic for which you want to generate facts, Y is the actual output that we want to generate, and Z are the key facts that we would like to generate.

So far, we have seen advanced prompt patterns, how to construct them with illustrative examples and how you can use them across various diverse use cases. In the next Chapter we will discuss the various types of prompting and how they can change the way you interact with models.

3.6 Summary

- Advanced Prompt patterns are vital in crafting effective prompts that generate optimized and tailored responses. These patterns can be widely used in content generation and LLM-based applications.

- Outline Expansion Pattern involves creating a basic outline from your input, expanding on the selected sections in subsequent rounds, and then asking you what to expand next.
- Semantic Filter Pattern helps to filter out specific content and rewrite it without changing the meaning of the text. This pattern will be crucial in the content generation domain, where content is monitored and regulated.
- Refusal Breaker Pattern is prominent in generating alternative questions where language models cannot answer hypothetical questions.
- Cognitive Verifier Pattern helps generate additional questions on a given topic. The model then uses the responses to these questions to generate a coherent response.
- Flipped Interaction Pattern helps improve our knowledge on any specific topic. This pattern transforms the language model from a passive AI communicator to an active conversational partner.
- Infinite Generation Pattern allows unlimited content generation without explicit model guidance. At the same time, rules and guidelines can be embedded within the prompt to ensure topic, tone, and context consistency.
- Context Manager Pattern enables users to add, modify, or delete the context used by the language model to generate a response. This pattern helps tune the content as needed.
- Reflection Pattern is beneficial for developing content with logical reasoning and proper citations on a given subject.
- Fact Checklist Pattern is similar to the reflection pattern, but this pattern generates the content along with the key facts.

4 Prompting Techniques I

This chapter covers

- The fundamentals of prompting techniques with illustrated examples of using them to interact with models.
- Setting up LangChain on your local environment and associated libraries that we will use in this Chapter.
- A deep insight into prompting techniques and their workings and illustrated code examples of each technique with their results.
- The limitations that each of the techniques have and how they affect the model's performance.
- Ways to mitigate the limitations by leveraging one or more techniques and prompt patterns.

In Chapter 1, we discussed the basics of prompt engineering and LLMs and how they can affect prompt crafting. Chapters 2 and 3 discussed the core elements that form a prompt, basic and advanced prompt patterns, their use cases, and detailed examples of using them for real-world problems. In this chapter, we will discuss the fundamentals of prompting techniques and the basic types of prompt techniques, along with detailed examples of prompt patterns we learned in Chapters 2 and 3 that apply these prompt techniques.

Before delving into the specifics of prompting techniques, it's crucial to grasp the distinction between these techniques and the prompt patterns we've covered in Chapters 2 and 3. Prompt patterns, as the name suggests, provide reusable patterns for model interaction. Each pattern allows users to add context, examples, roles, etc., enabling the model to generate desired responses for specific scenarios. On the other hand, prompting techniques are how you can effectively interact with the model using the patterns we've previously discussed. Let's explore this distinction further with a few examples.

In the following example, we're instructing the model to assume the role of a graphic designer and explain the basics of graphic design. The prompt is as follows:

Can you act as a graphic designer and teach me the basics of graphic design?

The model then generates a response that explains the basics of graphic design, effectively helping us get started in the field.

Absolutely, I'd be happy to help you get started with graphic design! Here are some basics to get you going.

1. Understanding Design Principles

- **Balance:** Distributing elements evenly to create a sense of stability. This can be symmetrical or asymmetrical.
- **Contrast:** Using differences in color, shape, or size to make elements stand out.
- **Alignment:** Ensuring elements are visually connected and organized.
- **Repetition:** Repeating elements to create consistency and unity.
- **Proximity:** Grouping related items together to create a relationship between them.

In this example the prompt utilizes the Persona prompt pattern, where we direct the model to assume a persona. However, it also employs a technique called the zero-shot prompting technique (discussed further in Section 4.3).

Let's examine another example to further understand the concept of prompting techniques. The following example shows a prompt guiding the model to assume the user to be an aspiring political analyst.

Can you assume that I am an aspiring political analyst and teach me the basics of being a political analyst?

The model parses the prompt and generates a response that can help you get your career started as a political analyst.

Absolutely, I'd be happy to help you get started on your journey as a political analyst! Here are some basics to get you going.

Understanding Political Systems

1. **Types of Governments:** Learn about different forms of government such as democracies, autocracies, and monarchies.
2. **Political Theories:** Study key political theories and ideologies like liberalism, conservatism, socialism, and nationalism.

Research Skills

1. **Data Collection:** Gather data from reliable sources such as government reports, academic journals, and reputable news outlets.
2. **Data Analysis:** Use statistical tools and software to analyze political data. Familiarize yourself with programs like SPSS, R, or Python.

This example uses an Audience persona pattern to direct the model while leveraging the zero-shot prompting technique when interacting with it. Can you identify the difference between a technique and a pattern? In simple terms, a prompting technique is a way to use different prompt patterns to guide the model in generating responses for different use cases. Choosing the appropriate technique depends on the model's capabilities, the type of responses required, and sometimes the prompt pattern used.

In this Chapter, we will discuss how to use various prompting techniques with the different prompt patterns we have examined so far and how to use them programmatically.

4.1 Why use Prompting Techniques?

Prompting techniques guide models to generate relevant, authentic, fact- and data-based responses. Understanding and using prompting techniques effectively will help improve the quality of your prompts with examples and added contexts, the quality of responses from the model, and your ability to leverage the full power of the model. Let us look at the most important reasons to use prompting techniques.

- **Enhance Response Quality:** Effective prompting techniques can help the model generate more precise, accurate, and less ambiguous responses. Adding examples and additional context can ensure that the responses generated are directly related to the prompt.
- **Maximize Model Capabilities:** Techniques like ReAct and Chain-Of-Thought help break down complex problems into smaller parts and leverage the model's reasoning capabilities. Other techniques, such as one-shot and multi-shot prompting, help add examples and additional context to the prompt to guide the model in generating relevant and accurate responses.
- **Understand and Mitigate Limitations:** Techniques such as zero-shot and one-shot prompting help one understand a model's limitations. You can enhance your prompts with examples, additional contexts, and guide rails to mitigate these limitations.
- **Build Diverse Applications:** Different prompting techniques and strategies allow you to build applications leveraging the model's capabilities in specific areas of expertise.

Before we explore the different prompting techniques, let's explore LangChain and how to set it up for your Mac. We will use LangChain to interact with models and run different prompting techniques along with prompt patterns we learned in Chapters 2 and 3.

LangChain is an open-source framework aimed at helping developers build LLM-based applications. It provides tools and abstractions to help developers customize, improve accuracy, and make the information generated by LLMs more relevant. For more information on LangChain please refer to: <https://www.langchain.com/>

Refer to the Appendix for installation instructions for both Windows and Mac/Linux.

Now that we have installed LangChain, we will continue our journey to understand prompting techniques.

4.2 Prompting Techniques

So far, we have examined the difference between prompt patterns and techniques, examined a few examples that helped us understand the basic concept of a prompting technique, and learned why prompting techniques are essential in prompt engineering. Let us discuss the different prompting techniques, their uses, limitations, and mitigations, and provide illustrated code examples for each.

4.2.1 Zero-Shot, One-Shot, and Multi-Shot Prompting

You have probably used zero-shot and one-shot prompting techniques in your interactions with various models. Let's disambiguate the names to bring some clarity.

As the name suggests, zero-shot is a prompting technique that relies on just the prompt with no examples or context. This technique relies on the model's capability to parse the prompt and generate a response solely based on the dataset on which it has been trained and tuned.

Well, how does this work? Models are trained and tuned with diverse, large, and extensive datasets that enable them to comprehend the prompt, understand the context, and generate a response. However, the model may not be able to generate responses if the model is not tuned or trained on a specific dataset, does not have enough data to generate a response, or has been restricted by the model owners to respond to specific queries.

Similarly, one-shot prompting is a technique that relies on providing one example along with the prompt. In this case, the model depends on the example provided and the prompt to generate the desired responses. Now that we have understood each technique's meaning, how do they relate to your interactions with models? Let us examine two examples to refresh our memory of these prompting techniques. The following example shows how the zero-shot prompting technique interacts with a model. First, let's look at the prompt:

Name the highest mountain in the world and in North America.

The model's response is based on facts and shows how the model tries to continue the conversation.

The highest mountain in the world is Mount Everest, which stands at 8,848 meters (29,029 feet) above sea level.

In North America, the highest mountain is Denali (formerly known as Mount McKinley), with an elevation of 6190 meters (20,310 feet).

Are you interested in climbing or just curious about these peaks?

This is a prevalent example of interacting with models—one you use every day. This technique relies on the model's training and tuning dataset. Can you now guess what multi-shot prompting means? Multi-shot prompting means the method consists of multiple examples that help the model generate responses. By providing a diverse set of examples and additional context, multi-shot prompting guides the model to generate responses that are not limited in creativity, can handle multiple tasks, and avoid limited generalization.

Let us look at an example of multi-shot prompting. The following example shows how you can prompt a model with example prompts and responses as a part of the prompt. Let's look at the prompt first.

Example 1:

Input: "I love this movie! It's fantastic."

Sentiment: Positive

Example 2:

Input: "This is the worst experience I've ever had."

Sentiment: Negative

Example 3:

Input: "The movie was okay, not the best but not the worst."

Sentiment: Neutral

Now analyze this sentence:

Input: "The service was excellent, but the movie and the food were just average."

The model uses the examples in the prompt to understand how the response should be generated. The response is as follows:

Given the examples and the prompt, let's analyze the sentiment of the sentence:

****Input:**** "The service was excellent, but the movie and the food were just average."

Analysis:

- ****Positive Aspect:**** "The service was excellent" indicates a positive sentiment towards the service.
- ****Neutral Aspect:**** "The movie and the food were just average" suggests a neutral sentiment towards both the movie and the food.

Sentiment Conclusion:

The sentence contains both positive and neutral elements. The positive sentiment regarding the service is clear, while the neutral sentiment about the movie and the food balances the overall sentiment. Thus, the overall sentiment can be categorized as ****Neutral****.

Although the service receives praise, the neutral comments about the movie and the food result in an overall sentiment that is balanced and leans towards neutral.

As you can see in this example, a multi-shot prompt consists of multiple examples or contexts that guide the model on generating the response. Next, we will see some programming examples of how you can use each of these prompting techniques using LangChain.

Before we start, we need to set up the virtual environment. Refer to the Appendix to see how to activate the virtual environment based on the OS on your local environment.

Once the virtual environment is activated, we will open a file to write code for zero-shot prompting. Let's look at the commands associated with opening the file and the code for creating a zero-shot prompt using LangChain in listing 4.1. In Listing 4.1, we import the required LangChain libraries and then:

- Instantiate ChatOpenAI with the temperature set to 0 and the Open AI API key. We will discuss temperature more in future chapters, so you do not need to worry about it now.
- We initialize the chat agent with a zero-shot agent type, verbose set to true, and then pass a simple prompt "Name the tallest mountain in the world and North America".
- Setting verbose to True results in the output consists of how the model parses the inputs, figuring out the correct answers from its knowledge base, and creating the final answer based on the answers to the question's two parts.

Look carefully, and you will see the nuances of zero-shot prompting. We already know that zero-shot prompting does not use any examples or context. Therefore, with zero-shot prompting, the model faces a question with no guidance and, often, no response generation training. So, what does the model do in this case?

- It breaks down the problem and then looks for the answer to the sub-problems in its database. The model breaks the problem into "Tallest mountain in North America" and "Tallest mountain in the world."
- The model retrieves the answer for both parts and then combines them to generate an observation: "Tallest mountain in the world: Mount Everest. Tallest mountain in North America: Denali."

- In the final step the model combines the observations to generate the final response "The tallest mountain in the world is Mount Everest, and the tallest mountain in North America is Denali.

Listing 4.1 shows writing zero-shot prompting using LangChain.

```
from langchain.agents import AgentType, initialize_agent
from langchain_openai import ChatOpenAI
from langchain_community.tools import ShellTool

shell_tool = ShellTool()

llm = ChatOpenAI(temperature=0, openai_api_key="<<Open API Key>>") #A

shell_tool.description = shell_tool.description + f"args
{shell_tool.args}".replace("{", "{{").replace("}", "}")

self_ask_with_search = initialize_agent([shell_tool], llm,
agent=AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION, verbose=True)

self_ask_with_search.run("Name the tallest mountain in the world and in North
America") #B
```

Result:

```
> Entering new AgentExecutor chain... #C
Question: Name the tallest mountain in the world and in North America
Thought: I can use the terminal to find the answers to these questions.
Action:
...
{
  "action": "terminal",
  "action_input": "echo 'Tallest mountain in the world: '; echo 'Mount Everest';
echo 'Tallest mountain in North America: '; echo 'Denali'"
}
...

Executing command: #D
echo 'Tallest mountain in the world: '; echo 'Mount Everest'; echo 'Tallest
mountain in North America: '; echo 'Denali'

Observation: Tallest mountain in the world:
Mount Everest
Tallest mountain in North America:
```

Denali

Thought:I should combine the information about the tallest mountains in the world and in North America to provide the final answer.

Action:

```
...
{
  "action": "terminal",
  "action_input": "echo 'Tallest mountain in the world: Mount Everest'; echo
'Tallest mountain in North America: Denali'"
}
...
```

Executing command:

```
echo 'Tallest mountain in the world: Mount Everest'; echo 'Tallest mountain in
North America: Denali'
```

Observation: Tallest mountain in the world: Mount Everest

Tallest mountain in North America: Denali

Thought:Final Answer: The tallest mountain in the world is Mount Everest, and the tallest mountain in North America is Denali. #E

> Finished chain.

#A Set up the ChatOpenAI API with the API key and temperature value.

#B Initialize a zero-shot react agent and run it with the prompt.

#C Model's initial thought process of how it should decompose the problem.

#D Model solves the sub-problems.

#E Model combines the answers from the sub-problems and forms the final answer.

Note that the code in Listing 4.5 uses deprecated methods and tooling and may not be ideal for usage when creating applications. However, this code and the result generated are essential for understanding how the model parses the prompt and generates the desired output.

Note that the above code uses deprecated methods and tooling and may not be ideal for usage when creating applications. However, this code and the result generated are essential for understanding how the model parses the prompt and generates the desired output.

Now, let us look at a more straightforward code example for your interactions with models using LangChain. Remember that LangChain allows you to interact with various models. For our example, we have chosen to use the Open AI API. Listing 4.2 shows the same prompt sent to the model using LangChain APIs without the verbose mode. The example shows how simple it is to interact with models using Python and LangChain.

Listing 4.2 shows code for a zero-shot prompt using LangChain.

```

from langchain_openai import ChatOpenAI

openai_api_key = '<<Open AI API Key>>'

llm = ChatOpenAI(temperature=0, openai_api_key=openai_api_key)

messages = "Name the tallest mountain in the world and in North America"

response = llm(messages)

output = response.content

print(output)

```

Response:

The tallest mountain in the world is Mount Everest, located in the Himalayas on the border between Nepal and China. The tallest mountain in North America is Denali (formerly known as Mount McKinley), located in Alaska, United States.

However, the real power of programming lies in interacting with models at scale. Let us look at one more example of multi-shot prompting in listing 4.3.

In Listing 4.7, we are using the multi-shot prompting technique. In the example we:

- Define the examples that the model will use as guidance and define a prompt template with the input variables as "Input" and "Sentiment" as we see in the examples and then defines what the prompt is. The prompt is a combination of the examples and the actual statement which we would like the model to analyze.
- Define the FewShotPromptTemplate from LangChain with examples and prompt template.
- Initialize the StrOutputParser, which takes the model output and converts to string and invoke the language model with the prompt containing text that is the positive leaning but negative and neutral sentiments.

Listing 4.3 shows an example of multi-shot prompting.

```

from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI
from langchain_core.prompts.few_shot import FewShotPromptTemplate
from langchain_core.prompts.prompt import PromptTemplate

examples = [{

```

```

#A

```



```

        "Input": "I love this movie! It's fantastic.",
        "Sentiment": "Positive"
    },
    {
        "Input": "This is the worst experience I have ever had.",
        "Sentiment": "Negative"
    },
    {
        "Input": "The movie was okay, not the best but not the worst",
        "Sentiment": "Neutral"
    },
]

prompt_example = PromptTemplate(
    input_variables=["Input", "Sentiment"], template="Prompt:
{Input}\n{Sentiment}") #B

#print(prompt_example.format(**examples[0]))

prompt = FewShotPromptTemplate(
    examples = examples,
    example_prompt = prompt_example,
    suffix = "Prompt: {Input}",
    input_variables = ["Input"],)

openai_api_key = '<<Open AI API Key>>'

llm = ChatOpenAI(temperature=0, openai_api_key=openai_api_key)

output_parser = StrOutputParser()

chain = prompt | llm | output_parser #C

multi_shot_prompt = prompt.format(Input="Despite the occasional setbacks and
challenges, the team's progress has been remarkable. There were moments of
frustration, but the overall journey has been incredibly rewarding. It's not
always easy, but the sense of accomplishment makes it all worthwhile.")

print("-----Response from the model-----")
print(chain.invoke(multi_shot_prompt))

```

Result:

Prompt: I love this movie! It's fantastic.

Positive

Prompt: This is the worst experience I have ever had.

Negative

Prompt: The movie was okay, not the best but not the worst

Neutral

Prompt: Despite the occasional setbacks and challenges, the team's progress has been remarkable. There were moments of frustration, but the overall journey has been incredibly rewarding. It's not always easy, but the sense of accomplishment makes it all worthwhile.

-----Response from the model-----

Positive

#D

#A Define examples that will guide the model.

#B Create a prompt template with input variables.

#C Pass the examples into the `FewShotPromptTemplate` and initialize the `ChatOpenAI` API.

#D Model parses the examples and generates the response.

Finally, the model responds with the sentiment analysis: "Positive". There are a few things to note in this example. We use the `FewShotPromptTemplate`, which allows you to provide a few examples. This means you could use the same `FewShotPromptTemplate` for one-shot prompting. We also used a prompt template to format our prompt and signify what our inputs and sample responses would look like. We will discuss more about programmatic templates and their uses in the future Chapters.

Do you want to use the code in Listing 4.3 for one-shot prompting? You could also modify the template to your desired output format and then use that when prompting the model with your input.

Now that we have learned about zero-shot, one-shot, and multi-shot prompting, we will examine their use cases and their limitations and mitigations.

USE CASES

Depending on the topic for which you are prompting the model, you will use zero-shot, one-shot, and multi-shot prompting techniques in almost all scenarios. However, we can divide the uses of these prompting techniques into some unique categories. Let us look at the categories where zero-shot, one-shot and multi-shot prompting can be very useful.

Sentiment Analysis: Sentiment analysis is LLMs' earliest and most used use case. You can leverage any model using zero-shot prompting to analyze the sentiment of a given text or sentence. If you want the analysis to be more descriptive and follow a format or example then you can use one-shot or multi-shot prompting. This will only enhance the model's analysis and ensure that the sentiment analysis suits your desired output format and detail.

Free form Content Generation: Remember your college essay that took several attempts to get it right? Remember how it is sometimes challenging to start a creative pursuit because of the mental block on how to start? Models can help you with these problems. You can and probably are using zero-shot prompting to generate a storyline, the outline of an essay, article, or paper you have meant to write for a long time. These prompting techniques build on each other, so you can use zero-shot prompting to kick-start the content generation and then use the response as input and guide the model to generate more content, modify existing content, or transform existing content into something new. You can use this prompting technique for text summarization, shortening, expansion, outline generation, grammar, tonality correction, and more.

Language Translation: Today, models are trained and tuned in several different languages. The zero-shot prompting technique is the most helpful here, as it relies on the model's training dataset. You are already using this technique for translation, and you could enhance the model's responses by providing additional information on the type of translation needed. This will also lead to greater transparency from the model on how it is doing the translation.

Data Manipulation: You can combine zero, one, and multi-shot prompting to modify data. Imagine you have a CSV of data that you need to modify. You can use zero-shot prompting to prompt the model to change data based on your filter criteria. If you want the data to be changed following a specific format, then you can use the template pattern with a one-shot prompting technique to guide the model in following the template and then update the data in the CSV file.

Chatbot Development: Models and their capabilities of comprehending human interaction have led to the development of chatbots that use LLMs as the base. How do we test them? A chatbot should be able to comprehend, parse, and reply to conversations without any additional context. This is where you can extensively use zero-shot prompting. Imagine you have created a chatbot for your company that specializes in cybersecurity. The real test of the bot is to differentiate between viruses in the general and cybersecurity contexts. If the bot needs an example to do so, it is not working as intended. Similarly, in other use cases of chatbots, zero-shot prompting helps test and validate the responses.

Despite their extensive use cases, these prompting techniques have some limitations. Let's look at them and how you can mitigate them.

LIMITATIONS & MITIGATIONS

As with every technique we will discuss in this Chapter, there are some limitations. The most important is the model's training and fine-tuning limit zero-shot prompting, as the method does not involve providing additional context or examples. If the model has not been trained on a specific topic or cannot comprehend the prompt, then the response may be incorrect, or the model may refuse to generate a response, citing its inability to do so due to limited training or tuning. One-shot prompting depends on a single example to generate responses. This can lead to over-correction as the model has only one example based on which it needs to process the prompt and generate responses. On the same lines, a single example or context may lead the model to limit its response and hamper creativity. Although multi-shot prompting mitigates all limitations that we see in zero-shot, and one-shot prompting the problem arises with input constraints of models. Models generally have constraints on how much input text can be fed to the model. You can mitigate this by using API based interactions that give much more granular and wider control over the responses that a model can generate.

4.3 Chain of Thought Prompting

Chain-of-thought (CoT) prompting plays a crucial role in enhancing the reasoning of language models. This technique helps models break down complex problems into smaller steps, enabling them to allocate additional computation for issues that require multiple reasoning steps. It provides detailed insights into the model's behavior and offers a step-by-step problem-solving process. This technique yields better results with a few examples and helps the model understand the required format. Chain of thought prompting is widely used in building language model applications for tasks such as coding, math problems, commonsense, and symbolic reasoning.

Let's start with the basic Chain-of-Thought prompting technique, known as the zero-shot chain-of-thought prompting technique.

4.3.1 Zero-shot Chain of Thought Prompting

Zero-shot Chain of Thought (CoT) prompting is a method used in language models to enhance reasoning abilities. As we have seen before, the zero-shot technique requires no additional context or example to generate the response. However, with zero-shot CoT prompting, the model is instead encouraged to think through the problem step by step till the conclusion.

In Listing 4.4, the prompt response involves reasoning steps, which might vary from iteration to iteration and model, but the response doesn't capture or explain these reasoning steps. Detailing these steps can be useful; for example, these steps can help us to verify the accuracy of a response or can help us to better understand how a problem can be solved. Listing 4.4 shows an example using the zero-shot CoT prompting technique.

Listing 4.4 shows zero-shot and CoT prompting techniques.
Prompt:

Q: Mrs. Smith currently has 10 students in her class. She receives 3 new groups of students over the week. Each group contains 4 students. How many students does Mrs. Smith have in her class now?

A: Let's think step by step

Response:

Sure, let's break it down step by step:

1. Identify the initial number of students:

- Mrs. Smith starts with 10 students.

2. Determine the number of new groups:

- She receives 3 new groups.

3. Calculate the total number of new students:

- Each group has 4 students.
- Total new students = 3 groups * 4 students/group = 12 students.

4. Add the new students to the initial number of students:

- Total students = initial students + new students.
- Total students = 10 + 12 = 22.

Solution: Mrs. Smith now has 22 students in her class.

In the example in Listing 4.4, we instructed the model with a simple mathematics problem to generate the solution with a detailed reasoning process using the zero-shot CoT technique. So, what did it guide the model to do? Well, the zero-shot CoT prompting technique guides the model in breaking down the thought process behind the final result and illustrating it in the response. This helps improve the model's performance, increase transparency, and validate the model's thought process. This thought-based reasoning becomes very valuable when dealing with tasks that require logical reasoning or problem-solving. Now, let us try the same example programmatically using LangChain. Listing 4.5 shows the same example as we saw in Listing 4.4.

Listing 4.5 shows an example using zero-shot CoT programmatically.

```
from langchain_core.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage

# replace with your openai api key
llm = ChatOpenAI(temperature=0, model_name="gpt-4o", openai_api_key="<<Open AI
API key>>" )

prompt = """ Q: Mrs. Smith currently has 10 students in her class.
She receives 3 new groups of students over the week. Each group contains 4
```

```

students.
How many students does Mrs. Smith have in her class now?
A: Let's think step by step "" #A

messages = [SystemMessage(content=prompt), HumanMessage(content=prompt)]
resp = llm(messages)
output = resp.content
print(output) #B

```

Response:

Sure, let's break it down step by step:

1. Mrs. Smith currently has 10 students in her class.
2. She receives 3 new groups of students.
3. Each group contains 4 students.

First, we need to find out how many new students she receives in total:

$\backslash [3 \text{ groups} \times 4 \text{ students per group} = 12 \text{ students} \backslash]$

Next, we add the number of new students to the number of students she currently has:

$\backslash [10 \text{ current students} + 12 \text{ new students} = 22 \text{ students} \backslash]$

So, Mrs. Smith now has 22 students in her class.

#A Initialize the ChatOpenAI API and define the zero-shot CoT prompt.

#B Define the messages and call the model with the prompt.

In Listing 4.5, we use the LangChain package to interact with a language model to solve the math problem. We, firstly, initialize the ChatOpenAI object with a specified temperature, model name, and open API key. Then define the math problem as a prompt using the zero-shot CoT technique. We then prepare the prompt as two messages: a SystemMessage containing the initial prompt to set the context for the language model, which provides background information or instructions on how the model should behave and also sets the tone or reasoning structure for the model. HumanMessage represents the input from the user, simulating a real human interaction. These messages help guide the language model behavior, which processes them and generates detailed responses stored in resp.content.

There is a minor issue with this technique. Since we do not provide any example that can guide the model, the model can generate several reasoning steps that are slightly different in every iteration but arrive at the same conclusion. This may be ok when working with models for isolated problems. Still, if you want to scale this and work with models to solve many problems, you must provide an example that guides the model in generating responses following the guidelines. This will ensure that the model is consistent in the format and reasoning technique, increase transparency, and improve validation of each reasoning step and, in turn, the result. To achieve this, we should use the few-shot CoT prompting technique.

4.3.2 Few-shot Chain of Thought prompting

Few-shot Chain of Thought (CoT) prompting is an extension of the zero-shot CoT technique, where the model is prompted with a few examples of how to reason through a problem step by step before solving a new, similar problem. This technique can significantly improve the model's ability to handle complex tasks and multi-step reasoning problems. It also enables the model to follow a reasoning thought process consistently and improves validation capability. Let us now look at an example of the few-shot CoT prompting technique. Listing 4.6 shows a logical reasoning problem with few-shot CoT prompting technique.

Listing 4.6 shows a logical reasoning problem with few-shot CoT prompting.

Prompt:

Question: Mrs. Smith currently has 10 students in her class. She receives 3 new groups of students over the week. Each group contains 4 students. How many students does Mrs. Smith have in her class now?

Answer:

Total Students = Current Students + (Number of Groups * Students per Group)

Total Students = 10 + (3 * 4)

Total Students = 10 + 12

Total Students = 22

Mrs. Smith now has 22 students in her class.

Question: A farmer initially has 15 apple trees in his orchard. Over the spring, he plants 5 new rows of trees, with each row containing 3 trees. How many apple trees does the farmer have in his orchard now?

Response:

Total Students = Current Students + (Number of Groups * Students per Group)

Total Students = 15 + (2 * 5)

Total Students = 15 + 10

Total Students = 25

Mr. Johnson now has 25 students in his class.

In Listing 4.6, we provide an example of a problem and its reasoning steps, including the final result. We then prompt the model with the actual question, and the model generates a response following the example. You could enhance this example by adding more examples to the prompt. This will only strengthen the guidance to the model and ensure that it can generate responses. Now, let's try this same example programmatically using LangChain. Listing 4.7 shows the few-shot CoT prompting technique using LangChain.

Listing 4.7 shows a LangChain based few-shot CoT prompting technique.

```
from langchain_core.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage

# replace with your openai api key
llm = ChatOpenAI(temperature=0, model_name="gpt-4o", openai_api_key="<<Open AI
API key>>" )

prompt = """
Question: Mrs. Smith currently has 10 students in her class. She receives 3 new
groups of students over the week. Each group contains 4 students.
How many students does Mrs. Smith have in her class now?
Answer:
Total Students = Current Students + (Number of Groups * Students per Group)
Total Students = 10 + (3 * 4)
Total Students = 10 + 12
Total Students = 22
Mrs. Smith now has 22 students in her class.
Question: A farmer initially has 15 apple trees in his orchard. Over the spring,
he plants 5 new rows of trees, with each row containing 3 trees.
How many apple trees does the farmer have in his orchard now?
"""
#A

messages = [SystemMessage(content=prompt), HumanMessage(content=prompt)]
resp = llm(messages)
output = resp.content
print(output)
#B

Response:

To find out how many apple trees the farmer has in his orchard now, we can use
the following calculation:
```



```
Total Apple Trees = Initial Apple Trees + (Number of New Rows * Trees per Row)
```

Given:

- Initial Apple Trees = 15
- Number of New Rows = 5
- Trees per Row = 3

Now, plug in the values:

```
Total Apple Trees = 15 + (5 * 3)
```

```
Total Apple Trees = 15 + 15
```

```
Total Apple Trees = 30
```

The farmer now has 30 apple trees in his orchard.

#A Initialize the ChatOpenAI API and create the few shot prompt using an example.

#B Call the model with the message and then print the output.

In Listing 4.7, we use the LangChain package to interact with a language model to solve the math problem. We firstly, initialize the ChatOpenAI object with a specified temperature, model name, and open API key. We then define the math problem using the few-shot CoT technique. Then we prepare the prompt as two messages: a SystemMessage containing the initial prompt to set the context for the language model, which provides background information or instructions on how the model should behave and also sets the tone or reasoning structure for the model. HumanMessage represents the input from the user, simulating a real human interaction. These messages help guide the language model behavior, which processes them and generates detailed responses stored in resp.content. You can see the response from the model follows the example used in the prompt.

Now that we have learned about chain-of-thought prompting, we will examine their use cases their limitations, and mitigation

USE CASES

Chain-of-thought prompting can be essential to building LLM-based applications that improve productivity in various domains. Here are some of the most common use cases.

Coding Assistant: Embedding chain-of-thought prompting with proper examples helps train the model. It enables accurate response generation, which can assist developers in tracing the root causes of issues step by step and implementing new functionality in any programming language.

Content Generation: If you want to generate the content for a given topic in a specific format, you can use the chain-of-thought prompting technique to instruct the model, context, and instructions. Developing a wrapper on top of the language model using a chain-of-thought prompting technique and prompt patterns can become a handy assistant to editors and content developers in the media domain.

Hypothesis Testing: Hypothesis testing is crucial in every field, and it can be automated using a chain-of-thought technique with varied examples. This technique helps researchers conduct experiments and interpret results sequentially by considering all variables and controls.

LIMITATIONS & MITIGATIONS

Chain-of-thought prompting mitigates the limitations of zero-shot, one-shot, and few-shot prompt techniques. This technique is heavily utilized in popular prompt engineering frameworks and LLM-based applications. However, it needs to improve in a few cases, such as complex reasoning sequences heavily dependent on initial assumptions, bias, and error propagation. To overcome these limitations and improve accuracy, you should use reasoning chains, review and automate the initial assumptions with up-to-date information, and incorporate the feedback loops that allow the reasoning process.

4.4 Self-Consistency

Self-consistency is a technique used to enhance the reasoning abilities of large language models (LLMs). It involves generating multiple reasoning paths through techniques like chain-of-thought or few-shot prompting techniques and then selecting the most consistent answer. The process begins by initiating the prompt with a reasoning example using one of these techniques. The model then generates a diverse set of responses by running the same prompt multiple times, and the most consistent response is chosen from the outputs. This process will help the model to improve its performance and generate consistent and accurate responses.

Self-consistency is a sophisticated technique utilized in prompt engineering to enhance the performance of large language models, especially in commonsense reasoning, mathematics, coding, and statistical analysis.

Imagine you want to assess the sentiments as positive or negative in customer reviews for a chain of restaurants. You can pass the customer reviews dataset iteratively in the prompt to the language model and generate the sentiments. Still, you should validate the model's responses as you would if using a programming language to analyze the reviews. To address this issue, you can guide the model with a self-consistency prompting technique, which enables the model to request the same prompt multiple times and pick the most consistent response. This helps to improve the accuracy and validation of responses from the language model. Since it's hard to showcase complex datasets here, let us look at a few examples of the self-consistency prompting technique. The following example shows how you can use self-consistency for a logical reasoning problem. First, let's look at the prompt for self-consistency.

Q: A farmer has 20 apple trees. Each tree produces 10 apples per day. The farmer eats 4 apples himself and uses 6 apples to make pies every day. He sells the rest for \$0.50 per apple. How much money does he make every day?

A: Let's think step by step.

Iteration 1 Output:

The farmer has 20 apple trees, and each tree produces 10 apples per day, so he gets a total of 200 apples per day (20 trees * 10 apples).

He eats 4 apples and uses 6 apples to make pies, which totals 10 apples. This leaves him with 190 apples to sell each day (200 apples - 10 apples).

He sells each apple for \$0.50, so he makes \$95 per day (190 apples * \$0.50).

Iteration 2 Output:

The farmer produces a total of 200 apples per day (20 trees * 10 apples).

After eating 4 apples and using 6 apples for pies, he has 190 apples left to sell (200 apples - 10 apples).

At \$0.50 per apple, he makes \$95 per day (190 apples * \$0.50).

Iteration 3 Output:

The farmer has 20 apple trees, with each tree producing 10 apples per day, giving a total of 200 apples daily (20 trees x 10 apples).

Every day, the farmer consumes 4 apples and uses another 6 for pies, totaling 10 apples used daily (4 + 6).

This leaves him with 190 apples to sell each day (200 - 10). At \$0.50 per apple, he earns \$95 daily from apple sales (190 apples x \$0.50).

In this example, we directed the language model with simple prompts multiple times to generate diverse reasoning paths. In this example, the result is the same even though there are diverse reasoning paths. It should be reasonably simple to generate the most consistent response from the language model by asking the exact prompt multiple times. Imagine if you want to solve a series of reasoning problems and get the most consistent response; it's a time-consuming task. Instead, you need to develop an LLM-based application by incorporating the self-consistency technique, which helps to request the language model multiple times to generate a diverse set of reasoning paths and categorize the responses based on reasoning paths. Then, pick the most consistent response. To improve the response and format further, we can incorporate the feedback data to train the model. Now, let's try this programmatically using LangChain. Listing 4.8 shows the code to implement self-consistency technique.

Listing 4.8 shows the self-consistency prompting technique.

```
from langchain_core.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain_openai import ChatOpenAI
```

```
# Define the prompt template
```

```
#A
```

```

prompt_template = """
Q: {question}
"""

# Create the prompt object
prompt = PromptTemplate(template=prompt_template, input_variables=["question"])
#A

# replace with your openai api key
llm = ChatOpenAI(temperature=0, model_name="gpt-4o", openai_api_key="<<Open AI
API key>>" )
llm_chain = LLMChain(prompt=prompt, llm=llm)          #B

# Define the question
question = """A farmer has 20 apple trees. Each tree produces 10 apples per day.
The farmer eats 4 apples himself and uses 6 apples to make pies every day. He
sells the rest for $0.50 per apple.
How much money does he make every day?"""          #C

# Generate multiple reasoning paths
responses = []
for _ in range(5): # Generate 5 different responses
    response = llm_chain.run(question=question)
    responses.append(response.strip())                #C

# Aggregate the results and find the most consistent answer
from collections import Counter

# Extract final answers from the reasoning paths
final_answers = [resp for resp in responses]
most_common_answer = Counter(final_answers).most_common(1)[0][0] #D

# Display all responses and the most consistent answer
print("Responses:")
for i, resp in enumerate(responses, 1):
    print(f"Response {i}: {resp}")

print("\nMost Consistent Answer:", most_common_answer)    #E

```

Response:

Response 5: To determine how much money the farmer makes every day, we need to follow these steps:

1. Calculate the total number of apples produced per day.
2. Subtract the number of apples the farmer eats and uses for pies.
3. Calculate the revenue from selling the remaining apples.

Let's break it down:

1. ****Total number of apples produced per day:****
 - The farmer has 20 apple trees.
 - Each tree produces 10 apples per day.
 - Total apples produced per day = 20 trees * 10 apples/tree = 200 apples.
2. ****Number of apples the farmer eats and uses for pies:****
 - The farmer eats 4 apples per day.
 - The farmer uses 6 apples to make pies per day.
 - Total apples eaten and used for pies = 4 apples + 6 apples = 10 apples.
3. ****Number of apples available for sale:****
 - Total apples produced per day = 200 apples.
 - Apples eaten and used for pies = 10 apples.
 - Apples available for sale = 200 apples - 10 apples = 190 apples.
4. ****Revenue from selling the apples:****
 - The farmer sells the remaining apples for \$0.50 per apple.
 - Revenue per day = 190 apples * \$0.50/apple = \$95.

Therefore, the farmer makes \$95 every day from selling the apples.

Most Consistent Answer: To determine how much money the farmer makes every day, we need to follow these steps:

1. Calculate the total number of apples produced per day.
2. Subtract the number of apples the farmer eats and uses for pies.
3. Calculate the revenue from selling the remaining apples.

Let's break it down:

1. ****Total number of apples produced per day:****
 - The farmer has 20 apple trees.
 - Each tree produces 10 apples per day.
 - Total apples produced per day = 20 trees * 10 apples/tree = 200 apples.
2. ****Number of apples the farmer eats and uses for pies:****
 - The farmer eats 4 apples per day.
 - The farmer uses 6 apples to make pies per day.

```

- Total apples eaten and used for pies = 4 apples + 6 apples = 10 apples.

3. **Number of apples available for sale:**
- Total apples produced per day = 200 apples.
- Apples eaten and used for pies = 10 apples.
- Apples available for sale = 200 apples - 10 apples = 190 apples.

4. **Revenue from selling the apples:**
- The farmer sells the remaining apples for $0.50 per apple.
- Revenue per day = 190 apples * $0.50/apple = $95.

#A Use the LangChain PromptTemplate to create a structured format for the input.
#B The prompt is connected to the OpenAI model using the ChatOpenAI package and is configured with
temperature settings and an OpenAI API key.
#C We then invoke the model with the prompt in a for loop, which generates the reasoning paths five times.
#D These responses are appended to a list after removing leading and trailing whitespaces using the strip()
function.
#E We select the most common response using the most_common() function and display the most consistent
response.

```

In Listing 4.8, for brevity we did not paste all the results. However, feel free to run this code in your LangChain environment and check the results of every iteration.

Now that we have learned about self-consistency prompting, we will examine their use cases and their limitations and mitigations.

USE CASES

Self-consistency prompting is a valuable technique in cases where consistent responses are essential to the specific topic. Let's look at a few use cases across various domains.

Customer Support: The self-consistency prompting technique expands the use of consistent and accurate responses, which is helpful in applications that depend on automated responses, such as customer support applications in every industry.

Brand Messaging: Employ the self-consistency technique, brand vision, and principles on the language model to ensure a consistent brand voice and messaging across all marketing communications and materials. This can become a valuable tool for any brand across industries.

Systems Operation: Operating systems are crucial for the product team and require 24/7 support. This process can be semi-automated by creating a custom bot with a language model using product principles and a self-consistency technique, which ensures consistent operational procedures, such as deployments and backups.

Regulatory & Compliance: Suppose you want to utilize language models in the regulatory and compliance department. In that case, you can use self-consistency prompting to ensure that actions comply consistently with legal regulations and improve productivity. This technique is crucial in sectors where regulatory compliance is tightly monitored, such as finance, healthcare, and manufacturing.

LIMITATIONS & MITIGATIONS

The self-consistency prompting technique is a follow-up to the CoT technique, and it mitigates some of CoT's limitations, such as consistency and accuracy. However, it is essential to acknowledge its limitations, such as requirements for large amounts of data, over-standardization, lack of adaptability, resistance to change practices, and computational complexity. Let's dive deeper into limitations to understand the additional context around these and the mitigations to handle these limitations.

Imagine if you want to build an LLM-based application to analyze every chat conversation and generate the most consistent answer. You can utilize the react prompting technique, but it is necessary for the model to generate a wide range of reasoning paths and then select the most consistent and optimized response. This process requires a large volume of data to train the model, resulting in computational challenges. To mitigate this limitation, tune the model with conditional rules, standard procedures and implement periodic checks, which helps to minimize the iterations over time and reduce the volume of data.

Another limitation is the sensitivity of noise: If you notice the noise of chat conversation, clarity or conciseness is missing, where the customer communicated with an agent that the talky bear isn't working, but there is no actual proof that it's not working. In such event, that prompt lacks clarity or conciseness, it may produce inaccurate responses. To handle such limitations, apply conditional rules such as "trust the customer if he mentions that the product isn't working even after troubleshooting" and comprehensive audits such as whether the model is considering previous chat conversations or not to influence the output.

4.5 ReAct Prompting

ReAct Prompting is a technique that integrates action and reasoning, allowing us to learn to make more appropriate decisions with detailed reasoning paths. It also enables interaction with external sources to retrieve additional information that leads to factual and reliable responses. This technique helps the model to generate dynamic reasoning, allowing it to formulate, modify, and refine action plans based on the context. The method is akin to Chain of Thought (CoT) prompting, which utilizes reasoning chains of question-answers as examples, and this is mainly used for arithmetic, common sense reasoning, and coding-related problems. Nonetheless, limitations such as the inability to access external information or update its knowledge base can result in inaccurate facts and the perpetuation of errors.

Simply asking a question in the form of a prompt to the language model will generate the answer, and this is nothing but zero-shot prompting. If you want to double down on the answer for detailed reasoning steps that can justify the answer, you can utilize the zero-shot CoT prompting technique. To generate a similar response for every question, you can use the few-shot CoT prompting technique, which will help you generate the response with detailed reasoning steps but may lead to errors such as predicting weather information as this technique cannot access external information.

To handle the limitations and improve the response without any errors, you should use the ReAct prompting technique. In this technique, actions and reasoning are synergized to generate the response, along with the ability to access external information. This helps the model generate a factual response with detailed reasoning steps.

When troubleshooting issues within a software application, the conventional method involves tracing the issue by analyzing the log data to locate the piece of code causing the problem and then fixing it, which can be time-consuming. An alternative approach would be to store the log data in a database and establish a connection so that the language model can access the log data. Then, instruct the language model with a react prompting technique to find the trace of the issue, which helps the product team improve productivity and enhance your application operating style. Figure 4.1 shows the diagrammatic representation of the ReAct prompting technique.

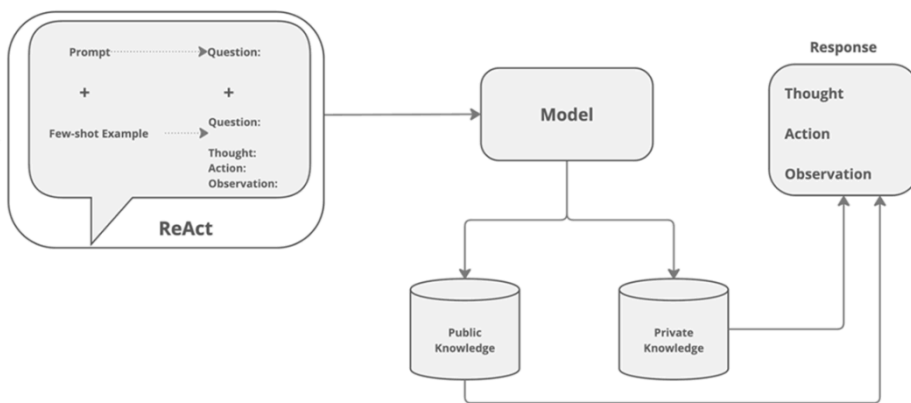


Figure 4.1 shows how the ReAct prompting technique works.

Let's analyze Figure 4.1. So far, we have discussed what ReAct prompting does. This diagram shows how to think about ReAct prompting. The technique involves a few-shot example that contains a sample thought on how to approach the problem, the action that follows the approach, and finally, the observation. The prompt and the few-shot example are fed to the model. The model parses the prompt and extracts data from publicly available references and its training data to generate the result. The result follows the same format as the example in the initial prompt. Let us now look at an example. The following example shows how the ReAct prompting technique works.

Question: What is causing the application to crash frequently?

Thought: I need to examine the software log data to identify any error messages or patterns that could indicate the root cause of the crashes.

Action: Search [application log data]

Observation: The log data contains multiple entries with timestamps, error codes, and messages.

Thought: I need to focus on the error messages and codes that appear around the times of the crashes.

Action: Filter [log data for error messages around crash times]

Observation: (Result 1 / 3) Error Code 500: Internal Server Error - NullPointerException at line 45 in Main.java

Observation: (Result 2 / 3) Error Code 500: Internal Server Error - NullPointerException at line 45 in Main.java

Observation: (Result 3 / 3) Error Code 500: Internal Server Error - NullPointerException at line 45 in Main.java

Thought: The repeated occurrence of a NullPointerException at line 45 in Main.java suggests a specific issue in the code.

Action: Lookup [NullPointerException at line 45 in Main.java]

Observation: (Result 1 / 1) The NullPointerException at line 45 in Main.java is caused by an attempt to access a method on a null object reference.

Thought: The root cause of the crashes is a NullPointerException due to a null object reference at line 45 in Main.java. This needs to be fixed in the code.

Action: Finish [NullPointerException at line 45 in Main.java causing crashes]

Question: Why is application performance slow during peak hours?

Isn't this an interesting problem for the model to solve? Let's look at the response from the model.

Thought: To determine why the application performance is slow during peak hours, I need to analyze various factors such as server load, database performance, network latency, and resource utilization during those times.

Action: Search [server load and resource utilization during peak hours]

Observation: The server load and resource utilization data show high CPU and memory usage during peak hours.

Thought: High CPU and memory usage could be contributing to the slow performance. I need to investigate further to identify specific processes or queries that are consuming excessive resources.

Action: Search [database performance and query logs during peak hours]

Observation: The database query logs indicate several long-running queries and high transaction volumes during peak hours.

Thought: Long-running queries and high transaction volumes can significantly impact application performance. Optimizing these queries and improving database indexing could help.

Action: Search [network latency and bandwidth usage during peak hours]

Observation: Network latency and bandwidth usage data show increased latency and high bandwidth consumption during peak hours.

Thought: Increased network latency and high bandwidth consumption can also contribute to slow application performance. Ensuring adequate network capacity and optimizing data transfer can help mitigate this issue.

Action: Summarize findings and recommend solutions

Observation: The application performance is slow during peak hours due to high CPU and memory usage, long-running database queries, high transaction volumes, and increased network latency and bandwidth consumption.

Recommendations:

...

4. Implement caching mechanisms to reduce database load.

5. Monitor and adjust resource allocation dynamically based on load patterns.

Action: Finish [Application performance slow during peak hours due to high resource usage, long-running queries, and network latency. Recommendations provided.]

Let's analyze this example. The prompt in this example is created with a one-shot example using the ReAct prompting technique. This technique helped the model generate a response by considering a sequence of thoughts and actions. The model ultimately concludes why the application is slow during peak hours by analyzing log data from the database. The model also recommends what you could do to mitigate the issue. Isn't seeing how the model can analyze a code error and generate recommendations interesting?

Let us look at another example. Suppose you want to understand the action and thought behind any general questions. We can craft the prompt with a one-shot example by incorporating the ReAct prompting technique, which guides the language model to respond to the questions in action and thought sequence till it reaches the final answer. Let's look at the prompt, and we incorporate the action and thought example into it.

Question: How deep is the Mariana Trench?

Thought: I need to search for the Mariana Trench, then identify the deepest point recorded in this trench.

Action: Search [Mariana Trench depth]

Observation: The Mariana Trench is the deepest part of the world's oceans.

Thought: The information is too general. I need specific data about the deepest point, known as the Challenger Deep.

Action: Lookup [Challenger Deep depth]

Observation: (Result 1 / 1) The Challenger Deep in the Mariana Trench reaches a depth of approximately 36,070 feet (10,994 meters).

Thought: I found the specific depth of the Challenger Deep. This is the information I was looking for.

Action: Finish [36,070 feet (10,994 meters)]

Question: What is the temperature range for the area that the Great Barrier Reef extends into?

Did you notice how we incorporated the thought in the prompt. The prompt guides the model in generating the following response.

Question: What is the temperature range for the area that the Great Barrier Reef extends into?

Thought: I need to search for the Great Barrier Reef, find the area it extends into, then find the temperature range of that area.

Action: Search [Great Barrier Reef]

Observation 1: The Great Barrier Reef is located off the coast of Queensland, Australia, in the Coral Sea.

Thought: It mentions the location of the Great Barrier Reef but not the specific temperature range. I need to look up the temperature range for this region.

Action: Search [Great Barrier Reef temperature range]

Observation 2: The temperature range of the Great Barrier Reef region typically varies from 73 to 86 degrees Fahrenheit (23 to 30 degrees Celsius).

Thought: I found the temperature range for the area where the Great Barrier Reef is located.

Action: Finish [73 to 86 degrees Fahrenheit (23 to 30 degrees Celsius)]

In this example, we used the ReAct prompting technique to assist the language model in generating a factual response supported by external evidence with action and thought process till it shows the temperature range of the Great Barrier Reef. Now, let's try doing this programmatically.

Listing 4.9 shows the ReAct prompting technique using LangChain. In the example, we define a prompt template. Defining the template helps in reusability across the program. Our definition is relatively simple and uses the variable "question" which we can replace with a prompt.

We then create a prompt template object using the template definition and assigning the input variables. We initialize the ChatOpenAI API with the API key, temperature, and model name and create an LLM Chain object with the prompt template object and model object(LLM)

Finally, we pass the prompt with the example to the LLM chain and then call the run method. The LLM chain object has the prompt template, which replaces the "question" input variable with our question and then runs the chain. Now, let us look at the code example.

Listing 4.9 uses LangChain to implement the ReAct prompting technique.

```
from langchain_core.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain_openai import ChatOpenAI

# Define the prompt template
prompt_template = """
Q: {question}
"""

# Create the prompt object
prompt = PromptTemplate(template=prompt_template, input_variables=["question"])
# replace with your openai api key
llm = ChatOpenAI(temperature=0, model_name="gpt-4o", openai_api_key="<<Open AI
API Key>>" )                                     #A

#Create an LLMChain object
llm_chain = LLMChain(prompt=prompt, llm=llm)        #B

# Define the question
question = "" Question: How deep is the Mariana Trench?
Thought: I need to search for the Mariana Trench, then identify the deepest point
recorded in this trench.
Action: Search [Mariana Trench depth]
Observation: The Mariana Trench is the deepest part of the world's oceans.
Thought: The information is too general. I need specific data about the deepest
point, known as the Challenger Deep.
Action: Lookup [Challenger Deep depth]
Observation: (Result 1 / 1) The Challenger Deep in the Mariana Trench reaches a
depth of approximately 36,070 feet (10,994 meters).
Thought: I found the specific depth of the Challenger Deep. This is the
```

information I was looking for.

Action: Finish [36,070 feet (10,994 meters)]

Question: What is the temperature range for the area that the Great Barrier Reef extends into?

"""

```
response = llm_chain.run(question=question)      #C
print(response)
```

Response:

Thought: I need to find information about the temperature range in the region where the Great Barrier Reef is located.

Action: Search [Great Barrier Reef temperature range]

Observation: The Great Barrier Reef is located in the Coral Sea, off the coast of Queensland, Australia. The temperature in this region varies depending on the season.

Thought: I need to find specific temperature data for the Great Barrier Reef area.

Action: Lookup [Great Barrier Reef seasonal temperature range]

Observation: (Result 1 / 1) The water temperature in the Great Barrier Reef typically ranges from 23°C (73°F) in the winter to 29°C (84°F) in the summer.

Thought: I found the specific temperature range for the Great Barrier Reef area. This is the information I was looking for.

Action: Finish [23°C (73°F) to 29°C (84°F)]

#A Define the prompt template, create the prompt template object, and then initialize the ChatOpenAI.

#B Create the model object.

#C Define the question, and then call the model object with the question.

In Listing 4.9, we use the LangChain PromptTemplate to create a structured format for the input. We initialize the ChatOpenAI package using the temperature setting, model_name, and OpenAI API key. Then, We use the LLMChain package to run the model with the prompt that incorporates the ReAct prompting technique, which helped to generate a response in thought and action sequence until it shows the temperature range of the Great Barrier Reef.

Now that we have learned about ReAct prompting, we will examine their use cases, their limitations, and mitigation

USE CASES

ReAct prompting responses or actions are influenced based on inputs. This approach is essential in a dynamic environment where rapid actions are needed. Here are a few use cases where react prompting is effectively used:

Live Chat Support: Using the react prompting technique with proper inputs will enhance and automate the chat support process, leading to daily improvement through feedback data input for training the model.

Market Trading: Instantaneous report generation and rapid action are essential in the trading ecosystem. This process can be streamlined by utilizing a language model, prompt patterns, and react prompting techniques to yield actions based on inputs. This approach is valuable in high-frequency trading, and other responsiveness can significantly impact the profitability of trading strategies.

Interactive Gaming: Training the language model based on previous feedback data and utilizing the react prompting technique will enhance player experience by providing instant feedback or changes based on player actions.

LIMITATIONS & MITIGATIONS

ReAct prompting is a powerful technique for quick and contextually appropriate responses with proper reasoning traces. However, several limitations, such as complexity, scalability challenges and resource intensive.

In Listing 4.9, react prompting generated the response with proper reasoning traces and actions for a given prompt, which helped us to understand the solution. Consider if you want to build an LLM-based application, such as a coding assistant where you manage reasoning traces and actions for every code. This process becomes tedious when you want to manage the reasoning traces and actions of every interaction by the developer, and this process may query external sources to perform actions that can be resource-intensive, potentially slowing down the response times. This process leads to scalability issues. To handle these limitations, you need to apply sophisticated rules such as validation techniques to manage reasoning traces and actions; these rules help to manage numerous queries and actions concurrently, leading to solve resource-intensive problems as well.

4.6 Prompt Chaining

Imagine a scenario where you are trying to solve a complicated mathematical problem. Intuitively, you will break the problem into subproblems and solve them individually. This applies to programming, mathematics, and any other topic you choose. At the core, breaking the problem down into subproblems helps in the thought process and ensures that there is a feedback mechanism for you to solve the larger problem. Similarly, the prompt chaining technique can achieve this in prompt engineering. Prompt chaining allows you to break tasks into subtasks, prompt the LLM with the subtasks, and use the responses from the subtasks as input into the other subtasks. This technique also helps improve the reliability and performance of LLMs as the subtasks enable the model to respond with accurate responses instead of a complex task in which the model may have difficulty generating responses.

You could also use the subtasks as a step to transform the output before it is used as input for the next subtask. This allows for transformation, validation, and verification of the output, increasing the model's reliability, controllability, and transparency. Prompt chaining also enables you to debug problems with responses, which may not be possible when using a single complex prompt. You could use the subtasks as steps to guide the model better and improve the responses.

Prompt chaining is very useful in building chatbots, creating conversational assistants, and creating content such as stories, essays, etc. Subtasks in prompt chaining will help you creating illustrative, engaging, and creative content while ensuring that the chatbots can be personalized with transformations and modifications to each subtask output. Figure 4.2 shows how to visualize prompt chaining.

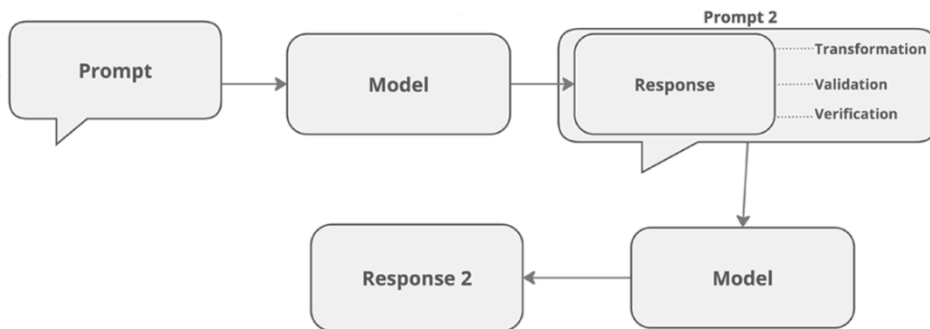


Figure 4.2 shows how prompt chaining technique works.

In Figure 4.2, we see how prompt chaining works. This technique breaks the problem into sub-problems and then prompts the model for each sub-problem. The figure shows it all works. We first send the model the first sub-problem, capture the response and transform the response, validate or verify the response, perform any other action as desired, and then prompt the model for the second sub-problem until we get our desired result. Let us now look at a few examples of the prompt chaining technique. The following example shows the prompt chaining technique. In this example, we want to generate a storyline. However, with creative processes like this, the story must be engaging and cohesive. To achieve this, we use prompt chaining. To get started, first, let's look at the prompt.

Can you generate a short story based on a plot narrative that I provide? Please generate a few lines and wait for my input and then generate the next part of the story. I will tell you when to stop. The characters are William, and Sarah. They are building their first home.

The model responds with the first response and then prompts the user for the next part of the story. Following is the first response from the model.

Of course! Let's start the story:

William and Sarah stood on the empty plot of land, their hearts brimming with excitement. The sun was setting, casting a golden hue over the rolling hills that would soon cradle their first home. With blueprints in hand and dreams in their eyes, they envisioned the life they would build together.

Please let me know when you're ready for the next part!

We then prompt the model with the next part of the story. Let's look at our prompt to guide the model in generating the next part of the story.

In the next part talk about the location and the plan for the home in their mind.

Now, let's look at the response from the model.

The location they had chosen was perfect—a serene spot on the outskirts of town, surrounded by lush greenery and a gentle stream that whispered through the landscape. They could already picture the mornings spent sipping coffee on the porch, listening to the birds sing.

In their minds, the house was a cozy two-story cottage with large windows to let in the natural light. The living room would have a stone fireplace, perfect for cold winter nights. The kitchen, Sarah's favorite part, would be spacious and modern, with an island in the center where they could cook together. Upstairs, they planned for three bedrooms, including a master suite with a balcony overlooking the hills.

Let me know when you're ready for the next part!

So, what did we learn from this example? Prompt chaining enables us to change the storyline as the model generates a story and ensures we can control the story's length, ending, and tonality. If you continue the conversation in the example, then you can change the story as it moves forward.

Let us take another example. Imagine you have a large dataset you would like to analyze. However, you need to figure out the end goal, which is dependent on the analysis as you go. You can upload the dataset to the model and guide the model in the data analysis. Let's look at an example of this. The following example shows data analysis using prompt chaining for data uploaded to the model using the web interface. In this example, we upload a dataset to the model and then prompt it to analyze the stock data.

Let's analyze this stock data.

The model analyzes the data and then generates the following response:

The stock data contains the following columns:

- **Date**: The date of the stock prices.
- **Open**: The stock's opening price on that day.
- **High**: The highest price the stock reached during the day.
- **Low**: The lowest price the stock reached during the day.
- **Close**: The stock's closing price on that day.
- **Volume**: The number of shares traded.

What specific analysis would you like to perform on this data? Examples include calculating averages, visualizing trends, or analyzing stock price volatility.

Next, we prompt the model with more questions on the dataset. Following the prompt:

Can you list the earliest stock date and the price?

The model generates the following response:

The earliest stock date in the dataset is **August 7, 2024**, with the following prices:

- **Open**: \$206.90
- **High**: \$213.64
- **Low**: \$206.39
- **Close**: \$209.82

Is there anything else you would like to explore or analyze?

We then prompt the model with a follow-up question on the stock data.

Can you tell me the volume of shares on the day when the share price was at its lowest?

The model's response is as follows:

On **August 7, 2024**, when the stock had its lowest price of **\$206.39**, the volume of shares traded was **63,516,422**.

Would you like to perform any further analysis on this dataset?

In this example, we uploaded a CSV containing Apple's stock data for one month, from 7 August 2024 to 7 September 2024. We couldn't show the uploaded CSV file for brevity, but you can upload a file via ChatGPT or any other model via its web interface. We use prompts to analyze the stock data successively and draw conclusions.

Now, let's try this programmatically. Listing 4.10 shows the code to upload and analyze a csv. Here, we use LangChain APIs to upload a CSV file containing Apple stock data and analyze it using the prompt chaining technique. Keep in mind that you should run this code through a sandboxed environment. Before running this code, you must install Pandas and Tabulate in your virtual environment.

Listing 4.10 shows the code to analyze a csv using LangChain APIs

```
from langchain.agents.agent_types import AgentType
from langchain_experimental.agents.agent_toolkits import create_csv_agent
from langchain_openai import ChatOpenAI, OpenAI

openai_api_key = '<Open AI API Key>'

llm = ChatOpenAI(temperature=0, openai_api_key=openai_api_key) #A

agent = create_csv_agent(
    llm,
    "AppleStock.csv",
    #verbose=True,
    allow_dangerous_code=True,
    agent_type=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
) #B

print(agent.invoke("How many rows are there in the csv file?")) #C

print(agent.invoke("Can you list the earliest stock date and the price?"))
#D

print(agent.invoke("Can you tell me the volume of share on the day when the share
price was at its lowest?")) #E
```

Response:

```
{'input': 'How many rows are there in the csv file?', 'output': 'There are 22 rows in the csv file.'}
```

```
{'input': 'Can you list the earliest stock date and the price?', 'output': 'The earliest stock date is 08/07/2024 and the corresponding price is 206.9.'}
```

```
{'input': 'Can you tell me the volume of share on the day when the share price was at its lowest?', 'output': '63,516,422'}
```

#A We first create an object of the model by initializing the ChatOpenAI with the API key and temperature.

#B Next, we create a zero-shot agent by initializing it with the path of the CSV from your local environment.

#C Invoke the agent with a prompt to the number of rows in the csv file.

#D Invoke the agent again to find the earliest stock date and the price.

#E Invoke the agent to get the volume of shares on the day when the share price was at its lowest.

You can use the same code to try parsing your own CSV file. However, you should keep in mind that the code should be run in a sandboxed environment and install Pandas and Tabulate for this code. You should also uncomment “verbose=True” and then re-run this code. You will get insights into how the model parses your prompts and the generates results. You can also try to add more prompts and play around with the data. Feel free to use your own custom CSV to see how the model parses it and the results it generates. If you want to explore more, then add validation/transform steps before the second prompt.

Now that we have learned about prompt chaining and also looked at code examples, we will examine their use cases and their limitations and mitigations.

USE CASES

Prompt chaining has vast uses and you can use it in combination with all the prompting techniques we have looked at so far. Let us look at some use cases.

Data Processing and Transformation: Similar to what we discussed in the use cases of previous prompting techniques, prompt chaining allows you to combine different types of prompting techniques to build a chain for your use case. Imagine you are analyzing a large dataset from a file or a database; you can use prompt chaining first to use zero-shot prompting to summarize the dataset, extract relevant data points, and then use CoT, few-shot CoT, or even ReAct prompting to modify the data, analyze the data and store the data in an external database.

Code Generation: Remember how you struggle sometimes to find the correct way of opening a file to read in Python or your favorite programming language? You search across many posts on Stackoverflow, GitHub, and other places to figure out what needs to be done. With prompt chaining, you can use the template pattern to define the coding guidelines, use zero-shot prompting to generate skeleton code, and then build towards the final solution with iterative prompting. Each step in between will allow you to validate, verify and then tune the program to your needs.

Creative Content Generation: Models, with their large training datasets, are more than capable of generating creative storylines, essays, and other text content. You can use prompt chaining to combine several different patterns, such as template, persona, and meta-language patterns, to create a sequence of patterns that help generate a storyline.

Customer Support: Have you interacted with chatbots when contacting customer support for your favorite service? The initial interaction is always very dry, lacks empathy, and almost feels robotic. This is because chatbots were not developed with LLMs. However, with LLMs, you can enhance how your chatbots can make the customer support experience more welcoming and human-like. Using prompt chaining, you can enable several reasoning paths to understand the user's issues, guide them through different paths to figure out the actual issue and link to guides.

Educational Tools: How has your experience been with online courses and education? Did you ever feel that the topic being explained was unclear, and you had to resort to different resources to clarify your understanding? Did that happen with books, too? What if there is a way to enhance that experience? Prompt chaining can help here immensely. Imagine you have an online course that you/your company sell to customers. A chatbot equipped with the course knowledge would be a great addition. You can enable the chatbot to answer users' questions about specific parts of the topic, leverage prompt chaining to assess the user's knowledge and generate personalized resources and exercises, gather feedback, and modify future learning steps.

POP QUIZ Do you want to try out how each of these techniques work? How about using prompt chaining or chain-of-thought prompting to analyze a dataset or application logs? You can use the sample code snippets in this Chapter as your starting point and create programs that can help you understand how each of these techniques work in the real world.

Now let us look at some limitations and mitigations of prompt chaining.

LIMITATIONS & MITIGATIONS

As with every technique we will discuss in this Chapter, there are some limitations. The main limitations of prompt chaining are:

- **Error propagation:** In Listing 4.10, we uploaded a CSV to analyze data. If the model gets erroneous data for the first prompt and we use the inaccurate data as input to the following prompt, the entire prompt chain will fail, as the result will be incorrect.
- **Increased Complexity:** The example in Listing 4.10 uses a reasonably small CSV for analysis. Even with this CSV, if your analysis is complex and one of the subtasks you want to analyze has a wrong or erroneous answer, it will become very complex for you to debug and find out what is wrong.

- **Prompt Effectiveness:** Model interactions depend on how clear, concise, and accurate your prompts are. If your prompts are vague, the model may generate vague responses that could pollute your subsequent prompts and the final output.
- **Computational Cost, Time, and Response Length Limitations:** Models generally have token limits, and the computational cost and time required to use them can be high as you use them for more complex use cases.

So, how do we overcome these limitations? Each step in your prompt chaining technique can have a validation step that ensures that the model can parse the data correctly before you prompt it for subsequent invokes. Similarly, to tackle complexity, you can use models specifically trained for statistical analysis, etc. The effectiveness of your prompts will play a significant role in your interaction with models. Therefore, you should practice and improve your prompts' effectiveness for more minor use cases and then use prompt chaining for complex use cases. To battle the computational cost and time required to work with the model, you can cache intermediate results so the chain does not have to start from scratch. You should also compare and contrast all the models the ROI you can expect when using the models for complex analysis and then choose the best option.

Now that we have looked at the basic prompting techniques, you can try them with various examples. Use different models to see how each responds to the techniques we have discussed. You will by using each of the techniques, understand which models need more examples vs which ones are very good at generating relevant responses. This will help you in choosing the appropriate model and technique along with the pattern to use with the model.

In the next Chapter, we will look at a few more prompting techniques that build upon the methods in this Chapter and the prompt patterns we have reviewed in Chapters 2, and 3.

4.7 Summary

- Prompting techniques play a crucial role in designing effective prompts in conjunction with prompt patterns to improve the response to different problem statements. Eventually, this helps build the most efficient LLM-based application in any domain or industry.
- Zero-shot prompting is a technique that instructs the language model to perform a task without any references or examples to guide it.
- One-shot prompting is a follow-up to zero-shot prompting. It instructs the language model to perform a task with a single example.
- Few-shot prompting allows the language model to perform a given task with multiple examples for an optimized response.
- The chain-of-thought prompting technique enables an accurate response from the language model by breaking down a complex dialogue into interleaved questions.

- ReAct prompting is a technique that combines reasoning and acting by prompting the model to generate the response with detailed reasoning steps and actions.
- Self-consistency prompting begins with generation of various reasoning paths for a given prompt and then determining the most consistent response.
- Prompt chaining helps break problems into sub-problems and then form a sequence of prompts of either the same or different patterns, allowing you to validate or transform the response before calling the following prompt in the sequence. This technique is helpful in complex uses of creative content generation, customer support, code generation, and making personalized educational tools.

5 Prompting Techniques II

This chapter covers

- A few more prompting techniques that build upon the ones we covered in Chapter 4 with illustrated examples of using them to interact with models.
- An insight into the newly released reasoning models, such as o3-mini from Open AI, Google's Flash thinking, and DeepSeek-R1.
- A deep insight into more complex prompting techniques and their workings and illustrated code examples of each technique with their results.
- The limitations of each technique and how they affect the model's performance.
- Ways to mitigate the limitations by leveraging one or more techniques and prompt patterns.

Chapter 4 discussed zero-shot, few-shot, multi-shot, Chain-of-thought, and ReAct prompting techniques. We also covered prompt chaining, which enables you to chain several prompts to break down a complex problem into smaller problems and add validation/verification/transformation steps between each prompt to achieve the end goal. In this Chapter, we will discuss a few complex prompting techniques that deal with text and video and build upon the methods we have already seen. We will leverage LangChain to build examples that closely relate to real-world applications and showcase how to use each technique to solve real-world use cases.

So far, we have examined Chain of Thought prompting using zero-shot, one-shot, and multi-shot techniques. In this section, we will first look at the latest reasoning models from Open AI, Google, and DeepSeek and compare their reasoning capabilities to those of generic models. We will also discuss Automatic and Tree-of-Thought prompting techniques more suited for custom use cases where generic reasoning models may not perform well because of a lack of examples.

5.1 Open AI's O3-mini Reasoning Model

Open AI was the first to market with ChatGPT's model and interface. It recently released Open AI o3-mini, a reasoning model capable of solving mathematical problems with detailed reasoning steps. O3-mini is accessible via ChatGPT's web interface. You can enter your prompt and use the "Reason" selection to process the prompt. Clicking "Reason" will activate o3-mini and generate reasoning steps for your problem. Let us take a simple example and see how o3-mini generates the reasoning. Figure 5.1 shows ChatGPT's the modified web interface with a way to select "Reasoning".

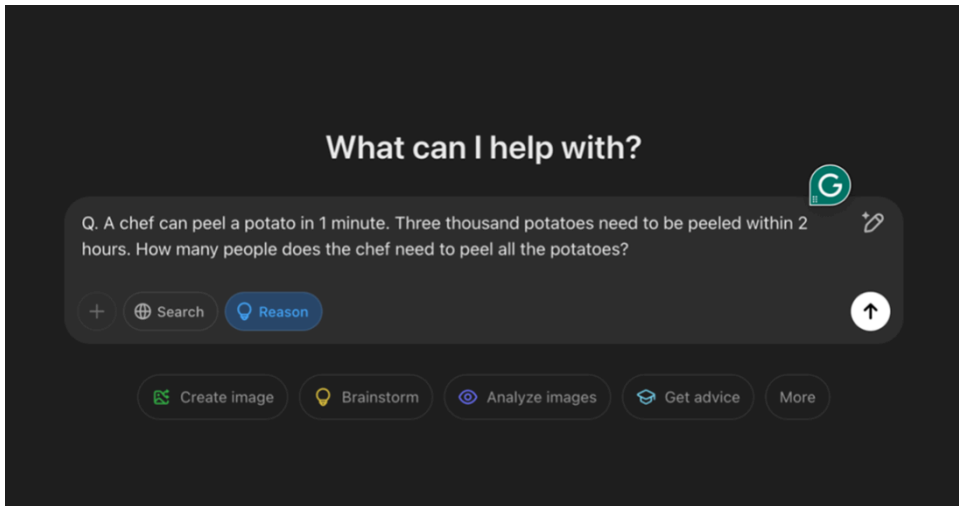


Figure 5.1

Figure 5.1 shows a sample mathematical problem and the "Reason" option selected using ChatGPT's web interface.

The model generates the response with detailed reasoning steps and the final answer.



Step-by-Step Explanation:

Determine the rate for one person: A chef can peel 1 potato per minute. Therefore, in 2 hours (which is 120 minutes), one person can peel:

$$1 \text{ potato/minute} \times 120 \text{ minutes} = 120 \text{ potatoes}$$

Calculate the number of people needed: To peel 3000 potatoes, if one person can peel 120 potatoes, the number of people required is:

$$\frac{3000 \text{ potatoes}}{120 \text{ potatoes/person}} = 25 \text{ people}$$

Final Answer:

The chef needs 25 people (including himself, if he is working) to peel 3000 potatoes in 2 hours.

The response from O3-mini shows the reasoning with detailed steps and the final result. Let's try this with a more complex example and see how it works. In this example, we prompt the model to help divide and decide whether a particular situation is possible or not. To do so we use the following prompt.

A group of five friends are trying to divide a sum of \$1087 amongst themselves, with each person only having whole dollar amounts. If Dolores has \$180, is it possible that she has the most money of the five friends, given that everyone has full dollar amounts?

The problem is about distribution and validating if the situation presented is possible. This requires the model to reason on two planes of thought. One being looking at the possibility of distributing money with restrictions of using only whole dollar amounts and the second being the possibility of Dolores having the most amount of money post distribution. The response from the prompt is as follows:



Step 1. Dolores has \$180.

Step 2. For Dolores to have the most money, each of the other four friends must receive less than \$180. That means the maximum each of them can get is \$179.

Step 3. If each of the other four friends gets at most \$179, the maximum total they can receive is:

$$4 \times 179 = 716 \text{ dollars. } 4 \times 179 = 716 \text{ \texttt{\{ dollars\}}.}$$

Step 4. Adding Dolores's share, the maximum total distribution under these conditions would be:

$$180 + 716 = 896 \text{ dollars. } 180 + 716 = 896 \text{ \texttt{\{ dollars\}}.}$$

Step 5. However, the total sum to be divided is \$1087. Since

$$896 < 1087, 896 < 1087,$$

there is a shortfall of

$$1087 - 896 = 191 \text{ dollars, } 1087 - 896 = 191 \text{ \texttt{\{ dollars\}},}$$

which means that at least one of the other friends must receive more than \$179. This would result in someone having as much as or more than \$180, so Dolores would no longer have the most money.

Final Answer:

No, it is not possible for Dolores to have \$180 and still have the most money among the five friends when dividing \$1087 into whole dollar amounts.

Now, let us look at DeepSeek-R1 and how it responds to the same questions.

5.2 DeepSeek-R1

DeepSeek-R1 is an advanced open-source LLM developed by the Chinese tech startup DeepSeek. DeepSeek-R1's reasoning matches that of humans as it explores more ways of thinking about the problem and validating its answers. Let's rerun the earlier examples with DeepSeek-R1 and see how it responds.

Q. A chef can peel a potato in 1 minute. Three thousand potatoes need to be peeled within 2 hours. How many people does the chef need to peel all the potatoes?

The response from DeepSeek is extensive, and it shows how it thinks about the problem before showing the results. The thought process is shown below followed by the response.



Okay, so the problem is: A chef can peel a potato in 1 minute. Three thousand potatoes need to be peeled within 2 hours. How many people does the chef need to peel all the potatoes?

First, let's understand the problem. We need to peel 3000 potatoes in 2 hours, and each chef can peel one potato in 1 minute. So, we need to figure out how many chefs are required to get all the potatoes peeled within that time frame.

Let me break it down. Let's start by converting the total time available into minutes because the peeling rate is given in minutes. Two hours is equal to 120 minutes (since 1 hour = 60 minutes, so 2 hours = $60 * 2 = 120$ minutes).

Now, each chef takes 1 minute to peel a potato. So, in 1 minute, one chef can peel 1 potato. Therefore, in 120 minutes, a single chef can peel 120 potatoes (1 potato per minute multiplied by 120 minutes).

But we need to peel 3000 potatoes. So, if one chef can peel 120 potatoes in 120 minutes, how many chefs do we need to peel 3000 potatoes?

To find the number of chefs required, we can divide the total number of potatoes by the number of potatoes one chef can peel in the given time. So, that would be 3000 potatoes divided by 120 potatoes per chef.

Let me calculate that. 3000 divided by 120. Hmm, 120 goes into 3000 how many times? Let's see, $120 * 25 = 3000$. Because $120 * 10 = 1200$, $120 * 20 = 2400$, and then $120 * 5 = 600$. So $2400 + 600 = 3000$. So that's 25. Therefore, 25 chefs would be needed.

Wait, let me verify that again. If 1 chef can do 120 potatoes in 120 minutes, then 25 chefs would do $25 * 120 = 3000$ potatoes. That seems correct. So 25 chefs.

Is there another way to think about this? Maybe in terms of rates. The rate of peeling is 1 potato per minute per chef. So, the total rate needed is 3000 potatoes divided by 120 minutes. That would give the required rate in potatoes per minute.

Calculating that, $3000 / 120 = 25$ potatoes per minute. Since each chef contributes 1 potato per minute, you would need 25 chefs to achieve that rate. So, that's another way to get the same answer, 25.

Hmm, both methods give 25. So that seems consistent. Let me check if there's any possible oversight here. For example, does the problem consider any setup time or rest time for the chefs? The problem states that a chef can peel a potato in 1 minute, but it doesn't mention any other factors. So, assuming that the chefs can work continuously without any breaks, and that there's no downtime between peeling each potato, the calculation holds.

Another thing to consider is whether the chefs can work simultaneously without interfering with each other. The problem doesn't mention any limitations on the number of chefs working together, so we can assume that adding more chefs doesn't reduce efficiency. So, if we have 25 chefs, each can work independently and peel one potato every minute, so together they can do 25 potatoes per minute, which over 120 minutes would be 3000 potatoes.

Alternatively, if we think in terms of total work required. The total work is 3000 potatoes * 1 minute per potato = 3000 chef-minutes. The total time available is 120 minutes, so the number of chefs needed is total chef-minutes divided by time available: $3000 / 120 = 25$. Yep, same answer.

So, all different approaches seem to confirm that 25 chefs are needed. Therefore, I think the answer is 25.

****Final Answer****

The chef needs $\boxed{25}$ people to peel all the potatoes within 2 hours.

Now, the model responds with the calculation leading to the final result. The response is as shown below:



First, convert the total time available into minutes:

$$2 \text{ hours} = 2 \times 60 = 120 \text{ minutes}$$

Each chef can peel 1 potato per minute. Therefore, in 120 minutes, one chef can peel:

$$1 \text{ potato/minute} \times 120 \text{ minutes} = 120 \text{ potatoes}$$

To find the number of chefs needed to peel 3000 potatoes, divide the total number of potatoes by the number of potatoes one chef can peel in 120 minutes:

$$3000 \text{ potatoes} / 120 \text{ potatoes/chef} = 25 \text{ chefs}$$

Another way to verify is by calculating the required peeling rate:

$$3000 \text{ potatoes} / 120 \text{ minutes} = 25 \text{ potatoes/minute}$$

Since each chef peels 1 potato per minute, 25 chefs are needed to achieve this rate.

Thus, the final answer is: 25.

5.3 Gemini 2.0 Flash Thinking

Google's Gemini 2.0 Flash Thinking is an experimental AI model designed to enhance reasoning capabilities and transparency by displaying its thought process. Like DeepSeek-R1 and O3-mini, this model also displays its deliberations, line of reasoning, and assumptions leading to the conclusion. Let's use the earlier problem of dividing money amongst a group of 5 friends with this model and see how it responds.

A group of five friends are trying to divide a sum of \$1087 amongst themselves, with each person only having whole dollar amounts. If Dolores has \$180, is it possible that she has the most money of the five friends, given that everyone has full dollar amounts?

Response from the model is a little different from what we have seen before, and it shows how Gemini thinks about the problem and then comes up with the solution. The response is as follows:



Final Answer: No.

To determine if it is possible for Dolores to have the most money, we assume that she does and check if this assumption leads to a contradiction. Let Dolores have $A_1 = 180$ dollars. Let the other four friends have A_2, A_3, A_4, A_5 dollars respectively.

If Dolores has the most money, then $A_1 \geq A_2, A_1 \geq A_3, A_1 \geq A_4,$ and $A_1 \geq A_5$. This means $180 \geq A_2, 180 \geq A_3, 180 \geq A_4,$ and $180 \geq A_5$. Since everyone has whole dollar amounts, A_2, A_3, A_4, A_5 are non-negative integers. The total sum of money is \$1087, so $A_1 + A_2 + A_3 + A_4 + A_5 = 1087$. Substituting $A_1 = 180$, we get $180 + A_2 + A_3 + A_4 + A_5 = 1087$. Subtracting 180 from both sides, we have $A_2 + A_3 + A_4 + A_5 = 1087 - 180 = 907$. Since $A_2 \leq 180, A_3 \leq 180, A_4 \leq 180, A_5 \leq 180$, the maximum possible value for the sum $A_2 + A_3 + A_4 + A_5$ is achieved when $A_2 = A_3 = A_4 = A_5 = 180$. In this case, the maximum possible sum is $A_2 + A_3 + A_4 + A_5 \leq 180 + 180 + 180 + 180 = 4 \times 180 = 720$. However, we require $A_2 + A_3 + A_4 + A_5 = 907$. Since $907 > 720$, it is impossible to find values for A_2, A_3, A_4, A_5 such that $A_2 + A_3 + A_4 + A_5 = 907$ and $A_2 \leq 180, A_3 \leq 180, A_4 \leq 180, A_5 \leq 180$. Therefore, it is impossible for Dolores to have the most money.

So, how do each of these compare to their generic versions? The primary difference is thought processing. If you use ChatGPT without “Reason” selected and run the same prompts, you will most likely see the same answer; however, it will lack the thought processing and deliberations you see when using the reasoning-specific version of the model.

Does this mean we do not need chain-of-thought prompting techniques anymore? Remember, most models, including almost all the publicly available models, are trained on generic data that is freely available to them. They can and will be able to solve most problems easily. However, prompting techniques such as chain-of-thought are helpful when dealing with complex issues that involve unique use cases or scenarios. E.g., if you are an attorney and would like to have the model analyze a few past cases and give recommendations on your current case based on evidence available, then you need to provide the model with examples of past cases that it can parse and make use of when looking at the new cases.

5.4 Advanced Chain of Thought Prompting

So far, we have examined Chain of Thought prompting using zero-shot, one-shot, and multi-shot techniques. In this section, we will build upon the Chain of Thought prompting technique to discuss Automatic, and Tree of Thought prompting. These techniques incorporate chain-of-thought prompting but do so while mitigating some of the issues in regular chain-of-thought prompting techniques.

5.4.1 Automatic Chain of Thought

The chain-of-thought prompting technique showcases the model's reasoning for a particular logical/mathematical problem. How is Automatic Chain of Thought prompting better than generic Chain of Thought prompting technique? Well, when using the Automatic Chain of Thought prompting technique, you will use pre-defined prompts that will help in improving accuracy. Chain of Thought prompting already enables reasoning that promotes transparency.

The reasoning helps promote transparency and ensures the user understands how the model arrived at the solution. However, with one-shot and multi-shot chain-of-thought prompting, you must manually create examples with the prompt. For instance, you want to use the chain of thought prompting technique in an educational setup. You want the model to generate detailed reasoning steps for problems. Students will use the model to validate their reasoning once they independently solve the problem. The subject could be vast and require you to manually create many examples with detailed reasoning steps. Also, how do you know if two examples are enough to guide the model or if you need to come up with more examples? Manual example creation is also error-prone. How do we solve this problem? Automatic chain of thought prompting can reduce the manual creation of examples while ensuring detailed reasoning steps following a standard format with greater accuracy.

The automatic chain of thought prompting technique involves context preparation before the actual prompt and model interaction. Are you wondering why we need a dedicated context preparation step before we use the model to generate a response for our use cases? Let us step back and understand the limitations of one-shot and few-shot/multi-shot prompting techniques. In one-shot, few-shot/multi-shot prompting, we manually create examples that are used as context for the problem we want to solve by leveraging the model. Besides being error-prone, the bigger problem with this process is that bias creeps in quickly and affects the model's ability to generate unbiased responses for real-world use cases. The solution to this problem is greater diversity in the context that can effectively guide the model in generating unbiased, accurate, relevant, and standard-format responses. So, how does the automatic chain of thought prompting technique help with this?

Automatic chain of thought prompting technique involves the following steps:

1. **Collation:** Collation of sample prompts in the area/topic you will use the model for real-world use cases. The sample prompts are from a curated bank containing different prompt types and their responses.
2. **Clustering:** Cluster sample prompts manually or automatically using a clustering algorithm to form sub-groups of questions based on their relative relationship.
3. **Context Generation:** Generating responses to the sample prompts using the zero-shot technique by leveraging the "Let's think step by step" attached to each prompt.
4. **Contextual Prompting:** Collate the prompt and responses generated by the model from the context-generation phase and then prompt the model with the context and the prompt to create reasoning chains.

Figure 5.2 shows how automatic chain-of-thought prompting works. The four-step process ensures diversity in sampling, accuracy in sample creation, response generation, and relevancy of the solution to the prompts.

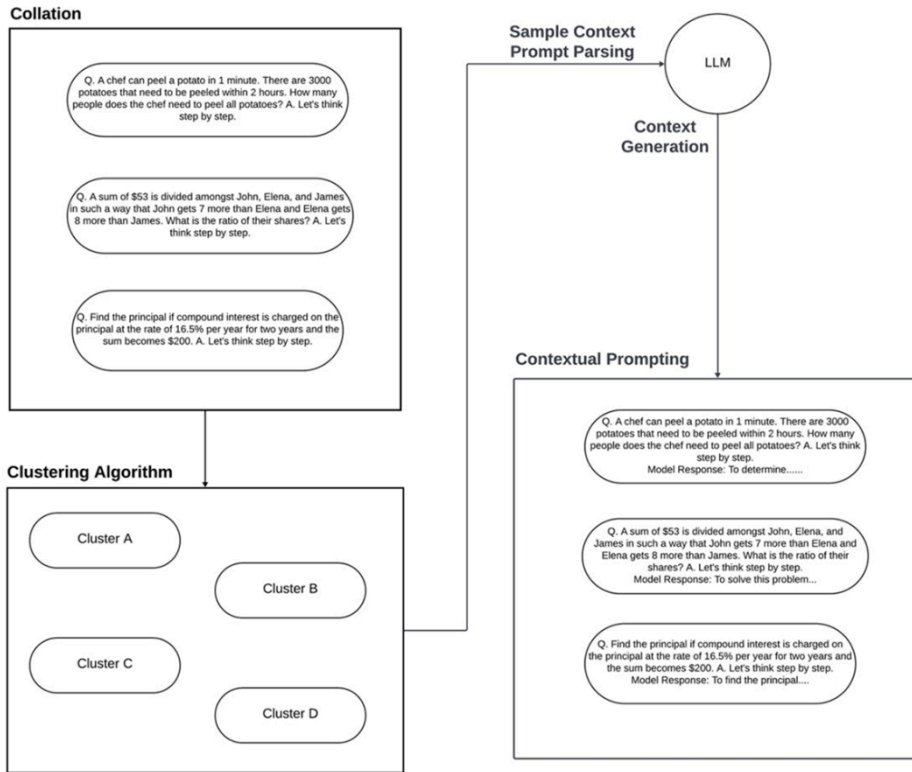


Figure 5.2 shows how Auto Chain-of-Thought prompting works.

Now that we've seen the steps, let's use a real-world use case to understand this process better. We all have struggled with learning a new concept or grasping a new topic at some point. We've explored videos, lectures, and notes to understand the concepts and contacted friends or other subject matter experts. For example, if you want to know how to solve a problem with detailed reasoning steps, you refer to resources where the same or similar problem has already been solved. However, a model trained or tuned in your specific field could shorten your research while providing reasoning chains for your problem. Now, think of this same problem at scale. If all students at school use the model for a specific topic, it could greatly benefit them because it provides quick turnaround times when they are stuck. This would shorten the learning curve considerably.

Let's take a more specific example. Imagine you are teaching students how to solve mathematical logical reasoning problems. You want the students to solve problems independently and then validate their solutions. To achieve this, you can leverage automatic chain-of-thought prompting. Let's take each step in the automatic chain of thought prompting technique and see how we can apply it to your problem space.

STEP 1: COLLATION

Collate questions that are good examples for the model to draw from. The following are some questions that we will use for our example.

Q. A chef can peel a potato in 1 minute. Three thousand potatoes need to be peeled within 2 hours. How many people does the chef need to peel all the potatoes?

Q. \$300 is divided amongst John, Elena, and James so that John gets seven more than Elena, and Elena receives eight more than James. What is the ratio of their shares?

Q. Find the principal if compound interest is charged on the principal at the rate of 16.5% per year for two years and the sum becomes \$200.

Q. 250 students took a mock test. 54% of the boys and 73% of the girls cleared the cutoff. If the total percentage of all students clearing the cutoff is 83%, how many girls appeared for the mock test?

In this example, we show four sample prompts. However, you could have many such prompts depending on your use case. These prompts could be from a curated prompt bank that contains prompts and their responses generated by the LLM. These samples can be as diverse as your use case. The Auto Chain of Thought prompting technique uses zero-shot prompting to gather reasoning chains for the sample questions. Therefore, we will add "Let's think step by step" to each sample question. Feel free to use a different phrase. The idea is to prompt the model to think in a detailed fashion and generate reasoning chains that help explain how the model computed the solution. The following are modified sample questions with the prompt to guide the model in generating reasoning chains.

Q. A chef can peel a potato in 1 minute. Three thousand potatoes need to be peeled within 2 hours. How many people does the chef need to peel all the potatoes?

A. Let's think step by step.

Q. \$300 is divided amongst John, Elena, and James so that John gets seven more than Elena, and Elena receives eight more than James. What is the ratio of their shares?

A. Let's think step by step.

Q. Find the principal if compound interest is charged on the principal at the rate of 16.5% per year for two years and the sum becomes \$200.

A. Let's think step by step.

Q. 250 students took a mock test. 50% of the boys and 70% of the girls cleared the cutoff. If the total percentage of all students clearing the cutoff is 80%, how many girls appeared for the mock test?

A. Let's think step by step.

In this step, we added a prompt to guide the model in generating reasoning paths and explaining how it calculated the answer to the problem. In the next step, we will first look at the clustering or categorization of these questions.

STEP 2: CLUSTERING

Cluster questions manually or using an algorithm. In our example, all questions pertain to mathematical logical reasoning. However, it will be helpful to cluster them according to their subtopics, such as accounting, basic math, ratio-based questions, etc. This process will increase the diversity of questions and help you judge the subtopics that have less representation and could lead to biased responses from the model. For the sake of brevity, we will not discuss any machine learning algorithm, such as k-cluster, etc. Feel free to apply any clustering algorithm you deem fit in this step.

STEP 3: CONTEXT GENERATION

This step will prompt the model with the sample prompts we created and clustered earlier. If you run these prompts manually, you can pick the samples equally from each cluster. You can also write code to pick a sample prompt from each cluster. The main goal is to run these prompts through the model without additional context and record the response. The prompt and its response collected in this step will be a contextual guide to the prompt in step 4. We see one of the prompts and its response from the model.

Q. \$300 is divided amongst John, Elena, and James so that John gets seven more than Elena, and Elena receives eight more than James. What is the ratio of their shares?

A. Let's think step by step.

The response:

Let's solve this step by step.

1. ****Assign Variables****:

- Let James' share be (x) .
- Elena's share is 8 more than James', so Elena's share is $(x + 8)$.
- John's share is 7 more than Elena's, so John's share is $((x + 8) + 7 = x + 15)$.

2. ****Total Share****:

The total amount shared is \$300. Therefore, the sum of their shares is:

$$x + (x + 8) + (x + 15) = 300$$

3. ****Simplify the Equation****:

Combine like terms:

$$3x + 23 = 300$$

4. ****Solve for (x) ****:

Subtract 23 from both sides:

$$3x = 300 - 23$$

$$3x = 277$$

Divide by 3:

$$x = \frac{277}{3} = 92.33$$

So, James' share is approximately \$92.33.

5. ****Calculate Elena and John's Shares****:

- Elena's share is $(x + 8 = 92.33 + 8 = 100.33)$.
- John's share is $(x + 15 = 92.33 + 15 = 107.33)$.

6. ****Ratio of their Shares****:

The shares of John, Elena, and James are approximately:

- John: \$107.33
- Elena: \$100.33
- James: \$92.33

The ratio of their shares is approximately:

107.33 : 100.33 : 92.33

To simplify, you can express these values as a ratio:

{John : Elena : James} = 107.33 : 100.33 : 92.33 \approx 107 : 100 : 92

Thus, the ratio of their shares is approximately **107:100:92**.

The highest mountain in the world is Mount Everest, which stands at 8,848 meters (29,029 feet) above sea level.

In North America, the highest mountain is Denali (formerly known as Mount McKinley), with an elevation of 6190 meters (20,310 feet).

Are you interested in climbing or just curious about these peaks?

The next step is to collect all the sample questions and reasoning chains and add them as context to the prompt.

STEP 4: CONTEXTUAL PROMPTING

In this step, we will use the sample prompts and their responses generated from the model as additional context and then prompt the model with the problem we want to solve. The model will use the context as guidance and generate a reasoning chain for the problem asked. The following shows the prompt with the prompt and the model's response.

I have a problem to be solved:

Q. \$1550 is lent out in two parts, one at 8% and the other at 6%. If the total annual income is \$106, find the money lent out at each rate.

Here is how I would like you to solve the problem:

Q. \$300 is divided amongst John, Elena, and James so that John gets seven more than Elena, and Elena receives eight more than James. What is the ratio of their shares?

A. Let's think step by step

<<insert the response from Step 3>>

The response:

Let's solve this step by step.

1. ****Assign Variables****:

- Let the amount lent at 8% be (x) .
- The remaining amount lent at 6% will be $(1550 - x)$ (since the total lent is \$1550).

2. ****Set Up the Equation for Interest****:

- The interest from the amount lent at 8% is $(0.08x)$.
- The interest from the amount lent at 6% is $(0.06(1550 - x))$.
- The total annual income from the interest is \$106, so we can write the equation:

$$0.08x + 0.06(1550 - x) = 106$$

3. ****Simplify the Equation****:

Distribute (0.06) to the terms inside the parentheses:

$$0.08x + 0.06 \times 1550 - 0.06x = 106$$

$$0.08x + 93 - 0.06x = 106$$

Combine like terms:

$$(0.08x - 0.06x) + 93 = 106$$

$$0.02x + 93 = 106$$

4. ****Solve for (x) ****:

Subtract 93 from both sides:

$$0.02x = 106 - 93$$

$$0.02x = 13$$

Divide by 0.02:

$$x = \frac{13}{0.02} = 650$$

So, the amount lent at 8% is $(x = 650)$.

5. **Calculate the Amount Lent at 6%:**

The amount lent at 6% is $(1550 - 650 = 900)$.

Final Answer:

- Amount lent at 8% = \$650

- Amount lent at 6% = \$900

In this example, we see the reasoning chain for the prompt. You can use this at scale for real-world use cases. The most significant advantage of this prompting technique is the elimination of manual context design, which eliminates errors and shortens the time from creation, ideation, and implementation. We will now leverage LangChain to implement the auto chain of thought prompting technique for the same example set we have seen earlier. In Listing 5.1, we see the programmatic implementation using LangChain.

Listing 5.1 shows leveraging LangChain to get response for all prompts.

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage

llm = ChatOpenAI(temperature=0, model_name="gpt-4o", openai_api_key="<<OpenAI API
Key>>")

#Read the file containing prompts
f = open('auto-cot-prompt.txt', 'r')
lines = f.read() #A

prompts = lines.split("\n\n")
print(len(prompts))
f.close() #B

f = open('auto-cot-response.txt', 'a') #C
responses = []
for prompt in prompts:
    message = [SystemMessage(content=prompt), HumanMessage(content=prompt)]
    llm_response = llm(message)
    response = llm_response.content
    f.writelines(prompt+"\n")
    f.writelines("\n")
    f.writelines(response+"\n")
    f.writelines("\n")
    responses.append(response)
    print(response)

f.close() #D
```

#A Set the ChatOpenAI API key and read the file containing prompts.

#B Split the lines from the file based on the next line separator.

#C Open a file to store the responses from the LLM.

#D Write the prompts and the responses to the file and close the file.

In Listing 5.1, we programmatically model Step 1 and Step 3 using LangChain and Python. For the example, we use the first three prompts from Step 2. We store these prompts in a text file, read the file in the program, and use the zero-shot prompting technique to prompt the model for a response. We then collect the responses and write them to another file, which we can use in Step 4. Listing 5.2 shows how we can do this programmatically using LangChain.

Listing 5.2 shows the Step 4 of Advanced Chain of Thought prompting.

```

from langchain_openai import ChatOpenAI

llm = ChatOpenAI(temperature=0, model_name="gpt-4o", openai_api_key="<<Open AI
API Key>> " )

with open("auto-cot-response.txt", "r") as f:
    examples = [line.strip() for line in f]

context = ""
context += " ".join(map(str, examples))                                #A

messages = "Q. $1550 is lent out in two parts, one at 8% and the other at 6%. If
the total annual income is $106, find the money lent out at each rate."

response = llm(context + "\n\n" + messages)                          #B

output = response.content

print(output)                                                         #C

```

#A Create an object of the ChatOpenAI API, open the auto-cot-response file we created in Listing 5.5 and combine all the lines to create a text blob containing the context for the prompt.

#B Call the llm object with the context and the prompt.

#C Print the response

Listing 5.2 shows how we can perform Step 4, combining the earlier LLM responses and prompting the model with a problem statement. Do you want to run this code and see how the model responds? The model follows the same response format as we saw in Listing 5.4. It details the step-by-step process it took to get to the solution. Do you now see the advantages of the Advanced CoT prompting technique? Feel free to try similar examples with the code and see how it works. Next, let's look at this prompting technique's use cases, limitations, and mitigations.

USE CASES

Advanced Chain-of-Thought prompting can be applied in all the areas where the Chain-of-Thought prompting technique can be applied. This technique's advantage over earlier versions is that it allows you to apply it at scale, reduces manual intervention, and allows you to control work with validation steps and clustering criteria that can push the model to respond with higher accuracy. Let us look at the use cases of this technique. Since this technique is best suited for at-scale problems, we will also look at their use cases that apply at scale. Clustering allows you to collate a diverse set of problems and cluster them into groups to ensure that you know the overall composition of the problem space. This helps remove bias from the model's solution to new problems and ensures that you can direct the model to generate responses that follow a specific format and are related to a specific topic. Scale is the key here. Large organizations can use this prompting technique to use the model for their use cases.

Training & Education: Imagine a company that has updated its security policy and wants all its employees to take the training. In a normal scenario, the company contacts its learning and development department to create a new training, update the affected policies, and then plan a rollout date. Announcements are then sent to employees to take the security training, and then slowly, the company gauges how many employees are proficient vs. how many need to retake the training to become proficient. Using this technique, the learning material, including the old policy, can be fed to the model, and then the model can be asked to generate a set of questions and their detailed answers based on the context provided, just like in Listing 5.4. Clustering and validation can be used to cluster the questions into subgroups, and then validation can be used to validate the answers at every step. Under training and education, the use cases are vast. This technique can also be used at schools to reinforce learning among students.

Task Planning & Execution: Since this technique follows a clear step-by-step process to show how it arrived at a particular solution, it can be used for planning & execution processes. Imagine you are creating a marketing strategy for a product at your company. You can use previous marketing strategies as prompts for the model, collect their responses, cluster them, validate them, and then prompt the model to create a strategy for your response.

Content Generation: Just like Chain-of-Thought prompting, you can use this technique to guide the model in generating or modifying content to your liking. With Advanced Chain-of-Thought prompting, you can first prompt the model to generate content based on your criteria, and then you can cluster them into groups that make the most sense. For content generation, ensuring you have a diverse set of examples would be great. The more diverse your examples, the better the content generation result.

LIMITATIONS & MITIGATIONS

Although this technique is better than the few-shot chain-of-thought prompting technique because it reduces the ability to introduce errors while creating examples for the model to understand, it still suffers from some limitations. The limitations of this start with the automatic generation of solutions for sample prompts. The model could generate faulty responses if the task is too complex or involves many steps. To mitigate this limitation, you can validate steps generated by the model and even break down the problem into smaller problems. In addition to retrieving prompt responses from the model, another limitation is the problem of maintaining a curated prompt and solution set. You could solve this by automating the creation of sample prompts and collating them. This will help generate clear and concise responses that you can use as data for your prompt. The second limitation is clustering the prompts into groups. An appropriate, tested, and validated clustering algorithm is recommended. This will reduce errors you may face while clustering and ensure you have diverse prompts to guide the model in generating appropriate responses.

5.4.2 Tree of Thought Prompting

The Tree of Thought (ToT) prompting technique mimics human brainstorming and problem-solving by generating multiple reasoning paths, evaluating them, and selecting the best outcome. Unlike standard Chain-of-Thought (CoT) prompting, which follows a linear reasoning path, ToT explores multiple thought branches. This allows models to perform more advanced reasoning at test time, dynamically adjusting their approach based on evolving problem constraints. It is ideally suited for solving mathematical reasoning problems, especially linear algebra, arithmetic, and logic. It's worth noting that these language models were not initially built to solve these problems. However, we have seen that they can solve complex problems effortlessly with the correct prompt. Therefore, it's unsurprising that the discipline of prompt engineering is such an exciting and popular one. The Tree of Thought (ToT) prompting aims to utilize the capabilities of language models to learn at inference time, simulating humans' brainstorming ability. In a brainstorming session, participants are encouraged to develop several candidate ideas to solve a problem and collectively evaluate different solutions to identify promising candidates. These ideas are refined and critiqued until the group arrives at an optimal solution. Therefore, in this technique, a 'thought' refers to an intermediate reasoning step or idea that contributes to exploring possible solutions. A 'solution,' on the other hand, is the outcome derived from evaluating and synthesizing these thoughts.

Imagine a tree. Each branch of the tree represents a potential reasoning path, and the model evaluates the viability of each path before proceeding to the next step. This test-time reasoning allows the model to iteratively refine its thought process, improving its accuracy and adaptability compared to traditional prompting techniques. That's the essence of the tree of thoughts prompting. This process is broken down into four steps: thought decomposition, thought generation, state evaluation, and search algorithm. Let's look at each of these steps and understand how they work together.

1. **Thought Decomposition:** Thought decomposition breaks down a problem into smaller manageable steps. This could be a few words, a line of an equation, or even a paragraph, depending on the complexity of the problem. These steps (thoughts) are evaluated individually or collectively to arrive at the outcome.
2. **Thought Generator:** The thought generator takes the current state of the problem and generates potential thoughts or next steps. The key is diversity and adaptability, as having a variety of potential solutions increases the likelihood of finding the best outcome.
3. **State Evaluator:** After generating potential thoughts, we need a way to evaluate them. This is where the state evaluator comes in, guiding us to determine which states or branches of thoughts are promising and should be explored further.
4. **Search Algorithm:** Finally, the search algorithm uses strategies like breadth-first or depth-first search, depending on the problem. These strategies determine the path the problem-solving process will take through the Tree of Thoughts.

With the Tree of Thoughts, we have a flexible, adaptable, and modular approach to problem-solving. Figure 5.3 shows a diagrammatic representation of how the Tree of Thoughts prompting technique works.

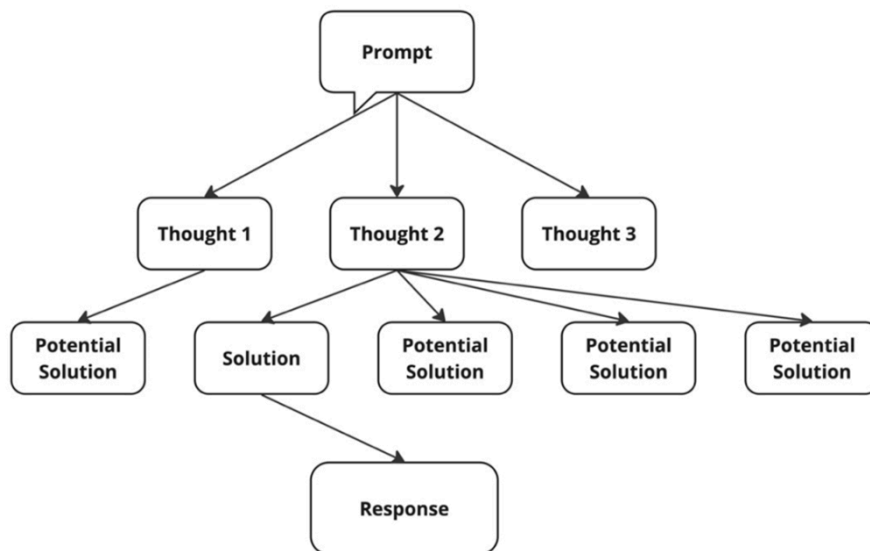


Figure 5.3 shows how Tree-of-Thought prompting works.

In Figure 5.3, we see how a model parses the prompt and then generates multiple thoughts. In Chapter 4, we learned about Chain-of-Thought prompting. Chain-of-thought prompting generates a linear reasoning chain that leads to the solution. Contrary to that, Tree-of-Thought prompting generates multiple reasoning chains. The model explores multiple thoughts or ideas that could solve the problem. Figure 5.3 shows how the model generates three thoughts for the prompt. Each of these thoughts results in a probable solution. Consider each of the thoughts as a node in a tree, and the potential solutions are branches. The model explores multiple branches at each decision point, like how branches grow on trees. If this process continues, the model could endlessly generate several branches. The model in the next step evaluates the branches and prunes the less promising or valuable ones. Although difficult to show diagrammatically, the Tree-of-Thought prompting technique allows the model to backtrack and re-evaluate previous branches if necessary. Models optimized for rapid and efficient reasoning, such as o1, o3, or those implementing 'flash thinking,' align particularly well with this technique by enabling quick exploration of thought branches. Let us look at an example to see how this prompting technique works. The following prompt guides the model in generating multiple thoughts of reasoning. Let us look at an example to see how this prompting technique works. The following prompt guides the model in generating multiple thoughts of reasoning.

Imagine three different experts are tasked with solving this mystery. Each expert will write down the first step of their reasoning and share it with the group. They will then evaluate their own logic and critique the reasoning provided by the other experts. The experts will use the principles of economics and behavioral psychology to refine their answers. At each step, if an expert identifies a flaw in their reasoning, they will trace the error back to its origin and adjust their approach. If an expert concludes their hypothesis is incorrect, they will explicitly state this and propose a new line of reasoning. The experts will assign a confidence level to their hypotheses at every step and continue until they agree on the most plausible explanation.

Scenario: A grocery store runs a promotional sale offering "Buy One, Get One Free" on a popular snack item. Despite the promotion, sales of the item remain unusually low during the sale period, which is unexpected based on prior promotions of this type.

Question: What is the most likely reason the promotion failed to increase sales?

Response:

Step 1: Initial Hypotheses

Expert A (Economist)

Hypothesis: The failure of the promotion to increase sales may be due to market saturation. Customers may already have a sufficient supply of the snack item at home, reducing their need to purchase more, even at a promotional price. Confidence Level: 70%

Expert B (Behavioral Psychologist)

Hypothesis: The promotion's framing might have confused or discouraged customers. For example, if the terms of the "Buy One, Get One Free" deal were unclear, or if the deal seemed too good to be true, customers might have hesitated to participate. Confidence Level: 60%

Expert C (Behavioral Economist)

Hypothesis: Customers may perceive the promotion as less valuable if the base price of the snack item was increased prior to the promotion. This could lead to a lack of trust or reduced perceived savings, resulting in lower sales. Confidence Level: 65%

Step 2: Evaluations and Critiques

Expert A evaluating others:

Expert B's hypothesis about framing confusion could be relevant if there is evidence that customers misinterpreted the terms. However, if other similar promotions in the store performed well, it may not hold.

Expert C's hypothesis about base price increases aligns with economic theory. If the promotion was perceived as a way to mask a price hike, this could discourage participation.

Expert B evaluating others:

Expert A's market saturation argument makes sense but assumes customers stockpile this item. If the snack is perishable or has a short shelf life, market saturation seems less plausible.

Expert C's price perception hypothesis is strong but assumes customers are aware of past prices. If this awareness isn't widespread, the effect could be weaker.

Expert C evaluating others:

Expert A's market saturation theory may be true for certain demographics but might not explain low sales across the board.

Expert B's framing confusion theory is plausible, particularly if marketing materials were unclear or mistrusted.

Step 3: Refinements

Expert A:

Flaw Identified: I assumed market saturation without considering the shelf life of the snack. If it's perishable, customers are less likely to stockpile.

New Hypothesis: Perhaps customers are substituting this snack with other similar items that were on sale during the same period.

Confidence Level: 60%

Expert B:

Flaw Identified: I overestimated the potential for confusion without direct evidence of unclear promotion materials.

New Hypothesis: Customers may have perceived the deal as unnecessary if they viewed the snack as unhealthy or indulgent, and this perception may have increased due to external factors like a health trend.

Confidence Level: 65%

Expert C:

Flaw Identified: I didn't account for whether customers were aware of past prices. If there was no significant price hike, my hypothesis loses strength.

New Hypothesis: Customers might have avoided the promotion because of social norms or stigma associated with appearing to "hoard" food items.

Confidence Level: 60%

Step 4: Convergence

The experts agree to combine insights for further analysis:

Market Saturation/Competition (A): If similar products were on sale, customers might have opted for alternatives.

Perception Issues (B): If the snack was seen as unhealthy, customers may have chosen not to buy it despite the deal.

Behavioral Norms (C): Social or psychological barriers to participating in the promotion (e.g., not wanting to hoard) could have contributed.

Step 5: Final Hypothesis

After reviewing store data, customer surveys, and promotional materials, the experts conclude:

Final Hypothesis: The promotion failed due to a combination of factors:

1. Competing products with better deals drew customers away.

2. A rising health-conscious trend reduced demand for the snack.
3. Customers perceived the deal as requiring them to purchase more than they needed, which conflicted with social or personal norms.

Confidence Level: 85%

The above example shows how the model can generate different reasoning paths and evaluate one against the other to create the final solution. The example prompts the model to assume three experts answer a logical reasoning question. Each of the experts critiques their and other's solutions. The model follows the guard rails defined and starts with each expert's hypothesis. The other experts then critique the hypotheses. Each hypothesis represents a new thought that could lead to a possible solution.

The critiquing mimics the "state evaluation" and iterative reasoning process of the model. In this step, the model evaluates the hypothesis to see if it could lead to a more likely solution. In the penultimate step, the model collates the hypotheses and the reasoning behind each solution and generates a final hypothesis. The final hypothesis generated by the model represents the consensus and the most likely solution to the core problem. Feel free to use the model of your choice and see for yourself how this technique works. Try with a text refinement problem. If you use Gemini, even when leveraging zero-shot prompting, i.e., without prompting the model to generate multiple reasoning chains, you can see the drafts the model created before generating the final response. This is how a model works internally using the Tree-of-Thought prompting technique. Now, let's see how this works programmatically using LangChain. Listing 5.3 shows the same example using LangChain. It is a lot simpler to do it programmatically!

Listing 5.3 shows Tree-of-Thought prompting using LangChain.

```

from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain_experimental.smart_llm import SmartLLMChain

question_part_1 = "Imagine three different experts are tasked with solving this
mystery. Each expert will write down the first step of their reasoning and share
it with the group. "
question_part_2 = "They will then evaluate their own logic and critique the
reasoning provided by the other experts. The experts will use the principles of
economics and behavioral psychology to refine their answers. "
question_part_3 = "At each step, if an expert identifies a flaw in their
reasoning, they will trace the error back to its origin and adjust their
approach. If an expert concludes their hypothesis is incorrect, they will
explicitly state this and propose a new line of reasoning. "
question_part_4 = "The experts will assign a confidence level to their hypotheses
at every step and continue until they agree on the most plausible explanation. "
question_part_5 = "Scenario: A grocery store runs a promotional sale offering
\"Buy One, Get One Free\" on a popular snack item. Despite the promotion, sales
of the item remain unusually low during the sale period, which is unexpected
based on prior promotions of this type. "
question_part_6 = "Question: What is the most likely reason the promotion failed
to increase sales? "
question = question_part_1 + question_part_2 + question_part_3 + question_part_4
+ question_part_5 + question_part_6
prompt = PromptTemplate.from_template(question)
llm = ChatOpenAI(temperature=0, model_name="gpt-4o", openai_api_key="<<Open AI
API Key>>")                                     #A

chain = SmartLLMChain(llm=llm, prompt=prompt, n_ideas=3, verbose=True)
chain.invoke({})                                #B

```

#A Create a prompt template from the question and create an Open AI object

#B Create a SmartLLMChain object with the number of ideas as 3 and then invoke the llm on the prompt.

To prompt the LLM to generate different thoughts, we use the SmartLLMChain library. This library enables us to provide the prompt with the number of ideas/thoughts we would like to be generated. In the example, we prompt the model to generate three thoughts. Try the same code with more "n_ideas" and see how the model generates thoughts, critiques, and the final resolution. Below, we see the response from running the program in Listing 5.2. The model follows the prompt, generates three thoughts, critiques each of the thoughts, and finally generates a response that it thinks is the most valuable and logical for the core problem.

Try running the program from Listing 5.7 with your desired prompt and check the response. You will be amazed to see how the model parses the prompt, generates thought paths, critiques the thoughts, and then arrives at the final solution. You could also add more contextual information to the prompt. The context you add will change the way the model finds the solution. Now, let's look at this prompting technique's use cases, limitations, and mitigations.

USE CASES

The Tree-of-thought prompting technique leverages the model's capabilities to explore the solution space and determine the most appropriate solution for the problem. Can you think of any possible use cases for such a prompting technique? The examples we saw earlier hint at using this example for logical reasoning problems. However, you can also use this for creative, logical interpretation scenarios. Let us go through a few of them.

Creative Writing: Are you writing an email, a document for your work or personal use, or refining a text? In all these cases, you sometimes run into issues where you do not know how to make the text sound or read better. This is where Tree-of-Thought prompting can help. You can leverage this prompting technique to refine the text. The biggest advantage is that the model shows you the reasoning behind refining or creating the text per your prompt and how it selected the final response. This is much better than the zero-shot, few-shot Chain-of-Thought prompting technique because the model can parallelly think of many probable solutions and compare the solutions to come up with a final solution.

Logical Reasoning: We already saw an example of logical reasoning in Listing 5.6. However, a well-trained model on specific topics could be used to create reasoning chains and perform logical analysis on related topics. Consider a model trained in corporate legal theory and use cases. You can leverage this model to evaluate cases and scenarios to see what reasoning paths it can develop. Similarly, you can use generally available models for math and logical reasoning problems.

Strategic Planning: Planning plays an important role in various projects across various industries. Have you encountered scenarios when you were working on a project and realized you missed an important milestone midway or have a blocker for which the solution is unclear? How did you approach that scenario? You probably brainstormed with your team to see what can be done. You can also use models in these scenarios. If the model has all the context related to the problem, it can generate reasoning paths you can analyze. For example, you want to create a plan to market a product that your company is making. You want to validate your marketing plan or create one. In this scenario, you can use the model to generate several marketing plans and their pros and cons. The model will also choose the most appropriate model for you based on the reasoning paths it generates.

Code Generation: When working on projects involving code, you can leverage the model and the Tree-of-thought prompting to generate different approaches to solve the problem. The model will generate multiple coding solutions and help evaluate the best-performing given the problem statement.

While these are some use cases where the Tree-of-Thought prompting technique can be very helpful, can you think of other use cases?

LIMITATIONS & MITIGATIONS

The Tree-of-thought prompting technique is a follow-up to the Chain of Thought (CoT) technique, and it mitigates some of CoT's limitations, such as accuracy and coherency. However, it is essential to acknowledge its limitations, such as the exponential growth of possible branches in complex problems, which are computationally expensive or time-consuming, ambiguous, and misinterpreting. Let's look at an example to understand these limitations better. Consider if you want to solve a quadratic equation where this equation has multiple ways to solve, such as factoring, using the quadratic formula, and completing the square. Each of these approaches represents a different branch. If the problem becomes more complex, the number of branches grows, making it harder to explore all possibilities. To handle these limitations, you must ensure that the model has guardrails in the form of additional context to the prompt to limit the branches and avoid computationally expensive solutions.

5.5 Generated Knowledge Prompting

Generated knowledge prompting is a technique that utilizes additional knowledge provided as part of the context to improve the performance of large language models, specifically in complex problems such as reasoning and coding. This technique begins with generating the knowledge based on the question and then incorporating the knowledge or information along with the original question to help the model make more accurate predictions. The final output is determined by selecting the prediction with the highest confidence.

Generated knowledge prompting is an effective method for integrating external knowledge into language models. This approach offers flexibility in addressing reasoning problems. In contrast, chain-of-thought prompting relies on internal or similar data association within the context and may not achieve the same level of accuracy. Before we go further, let's look at an example of how this prompting technique could enhance the model's response. The prompt below shows how you can test the knowledge of a model.

```
Question: "What will happen if you put a metal box in a microwave oven?"  
Answer choices: A) The box will melt B) The microwave will explode C)  
Nothing will happen D) The box will get hot
```

Response

The correct answer is not listed in the choices. Here's what will actually happen:

When you put a metal spoon in a microwave oven, it could cause electrical arcing, where sparks may jump between the spoon and other parts of the microwave. However, this is not the same as the microwave exploding or the spoon melting. The safest course is to avoid putting metal objects in a microwave as it could damage the appliance.

If forced to choose an option, none of the provided choices are completely accurate, but the closest is: C) Nothing will happen (though, in reality, sparks could occur).

In the above example, the response generated is accurate, but this may not always be true. Sometimes, the model can give the wrong answer based on incorrect assumptions, prompt parsing, etc. To improve accuracy, we could use the generated knowledge prompting technique. In this prompting technique, you generate the knowledge using a model for smaller sub-problems and then use that knowledge as context for the main problem in the model. This ensures that the model knows about different parts of the problem and can use it to solve the larger problem. Figure 5.4 shows a diagrammatic representation of this prompting technique.

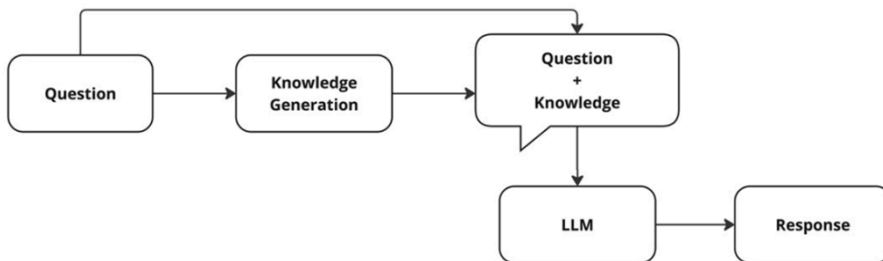


Figure 5.4 shows how the generated knowledge prompting technique works.

Figure 5.4 shows how the Generated Knowledge prompting technique works. In Chapter 4, we learned about Chain-of-Thought prompting, which generates the response based on the training dataset of the model. In contrast, Generated knowledge prompting follows multiple steps to solve problems. In the first step, we identify the context and problem statement based on the question, then prompt the model with the context to generate knowledge versions with as much detail as possible, provided with a few examples. Then, we craft the prompt, as shown in the figure, using the generated knowledge and the original question. The model will generate a response based on the prompt. This process is repeated several times to get more responses. Lastly, we select the prediction with the highest confidence. So why is it advantageous to generate knowledge for smaller problems leading up to the larger problem? If we skip this step, the model may hallucinate as it will be missing essential data required to solve the problem. With this technique, we fill the gap in cases where the model may not have such knowledge. Such scenarios are common in areas where accurate knowledge about intermediate steps is a must to solve the bigger problem. Now that we understand how the generated knowledge prompting technique works, let's look at an example prompt. We will walk through each step of the Generated Knowledge prompting technique. For our example, we will consider the problem of analyzing a patient's symptoms and diagnosing possible explanations for the symptoms.

WARNING The following steps contain healthcare examples. Please do not use models and prompt responses to make decisions about health issues.

Step 1: The process starts with gathering basic information about the scenario. In this case, the patient's symptoms and background are the core input for the model. The prompt below shows a question related to a patient's health. Suppose you prompt the model with the question it may or may not generate an accurate or comprehensive response. To solve this, we generate the knowledge about the symptoms and patient behavior and then input the knowledge along with the original question, as shown in steps 2 and 3.

Question: The patient is experiencing shortness of breath, chest pain, fatigue, and irregular heartbeat. The patient is also 55 years old and has a history of high blood pressure.

Step 2: This step generates relevant medical knowledge to enhance the prompt. The aim is to utilize factual, domain-specific information to help the language model deliver a more accurate and valuable response. The example below shows how you can generate knowledge related to a patient's symptoms. Listing 5.11 shows how knowledge is generated for the patient's symptoms.

Generate medical knowledge related to the symptoms listed for the patient. In patients experiencing shortness of breath, chest pain, and a history of high blood pressure, possible cardiovascular conditions include angina, heart failure, or arrhythmia.

Response

Patients with shortness of breath, chest pain, and a history of high blood pressure are at risk for several cardiovascular conditions, including:

1. Angina: Chest pain due to reduced blood flow to the heart muscle.
2. Heart Failure: The heart's inability to pump efficiently, causing fluid buildup, which can lead to shortness of breath.
3. Arrhythmia: Irregular heartbeats that may cause chest pain or shortness of breath.

These symptoms warrant prompt medical evaluation to prevent potential complications like a heart attack.

In this example, the model generates medical knowledge for each symptom the patient is experiencing. Similarly, knowledge related to patient behavior can be generated. Next, let's look at another example of generating knowledge for the patient's behavior.

Generate knowledge for patients over 50 years with these symptoms should also be evaluated for risks related to coronary artery disease and hypertensive heart disease.

Response

For patients over 50 years old experiencing shortness of breath, chest pain, and a history of high blood pressure, it is essential to evaluate for:

1. Coronary Artery Disease (CAD): Plaque buildup in the coronary arteries, reducing blood flow to the heart and increasing the risk of heart attacks.
2. Hypertensive Heart Disease: Long-term high blood pressure can lead to thickening of the heart muscle, impaired function, and heart failure.

Both conditions are common in older adults and require careful assessment to prevent serious cardiovascular events.

Step 3: The information collected in Step 2 is now combined into the final prompt that will be submitted to the language model. This enhanced prompt helps ensure that the system produces more accurate and relevant responses by considering the patient's symptoms and the relevant medical knowledge. Next, we see the final prompt that includes the knowledge and the original question.

Patients with shortness of breath, chest pain, and a history of high blood pressure are at risk for several cardiovascular conditions, including:

1. Angina: Chest pain due to reduced blood flow to the heart muscle.
2. Heart Failure: The heart's inability to pump efficiently, causing fluid buildup, which can lead to shortness of breath.
3. Arrhythmia: Irregular heartbeats that may cause chest pain or shortness of breath.

These symptoms warrant prompt medical evaluation to prevent potential complications like a heart attack.

For patients over 50 years old experiencing shortness of breath, chest pain, and a history of high blood pressure, it is essential to evaluate for:

1. Coronary Artery Disease (CAD): Plaque buildup in the coronary arteries, reducing blood flow to the heart, and increasing the risk of heart attacks.
 2. Hypertensive Heart Disease: Long-term high blood pressure can lead to thickening of the heart muscle, impaired function, and heart failure.
- Both conditions are common in older adults and require careful assessment to prevent serious cardiovascular events.

Question: Based on the patient's symptoms of shortness of breath, chest pain, and fatigue, along with a history of high blood pressure, and considering the potential risks of cardiovascular conditions such as angina, arrhythmia, or hypertensive heart disease, what is the most likely diagnosis and recommended next steps

Response

Based on the patient's symptoms—shortness of breath, chest pain, fatigue, and a history of high blood pressure—the most likely diagnosis could be Coronary Artery Disease (CAD) or Hypertensive Heart Disease. Both are common in older adults with a history of hypertension and can lead to serious complications if untreated.

Recommended next steps:

1. Immediate ECG (Electrocardiogram) to evaluate heart rhythm and potential ischemia.
2. Cardiac enzyme tests (e.g., troponin) to rule out a heart attack.
3. Echocardiogram to assess heart function and structure.
4. Stress test or coronary angiography to evaluate for CAD if clinically indicated.

These evaluations will help confirm the diagnosis and guide further treatment to prevent serious cardiovascular events.

In this example, the model generated the response based on the information provided about the original question. Consider this a use case for developing a medical diagnosis tool that assists doctors by suggesting potential diagnoses based on patient symptoms. This tool will improve doctors' efficiency and influence individual patients' prescribing behavior. We will now leverage LangChain to implement the Generated knowledge prompting technique for the same use case. Listing 5.4 shows the code example with LangChain.

Listing 5.4 shows code to add patient input and generated knowledge prompt.

```

from langchain.prompts import PromptTemplate
from langchain_openai import ChatOpenAI
from langchain.chains import LLMChain
import random

# Define the initial patient input (Step 1)
patient_input = {
    "age": 55,
    "gender": "male",
    "history": "high blood pressure",
    "symptoms": ["shortness of breath", "chest pain", "fatigue", "irregular
heartbeat"]
}
#A

# Step 2: Define the knowledge-augmented prompts
knowledge_prompts = [
    "Generate medical knowledge related to the symptoms listed for the patient. In
patients experiencing shortness of breath, chest pain, and a history of high
blood pressure, possible cardiovascular conditions include angina, heart failure,
or arrhythmia.",

    "Generate knowledge for patients over 50 years with these symptoms should also be
evaluated for risks related to coronary artery disease and hypertensive heart
disease."
]
#B

# Step 3: Create the final enriched prompt
enriched_prompt = f"""
A {patient_input['age']}-year-old {patient_input['gender']} with a history of
{patient_input['history']}
is experiencing {' '.join(patient_input['symptoms'])}.
{knowledge_prompts[0]}
{knowledge_prompts[1]}
What is the most likely diagnosis, and what are the recommended next steps?
"""
#C

```

#A Create the patient input consisting of symptoms, age, and other details.

#B Create the knowledge generation prompt with the symptoms of the patient.

#C Create final enriched prompt consisting of the patient inputs, generated knowledge and the question.

Listing 5.4 shows the code equivalent of Steps 1, 2, and 3 we discussed earlier. We create the patient input json, the knowledge generation prompt, and the final prompt that combines the patient input and the knowledge from the knowledge generation prompt. Listing 5.5 shows how we can leverage the Prompt Template and Python basics to generate multiple responses and pick the response with the highest confidence score.

Listing 5.5 shows code to generate responses and choose the best response.

```
# Initialize the OpenAI LLM
llm = ChatOpenAI(temperature=0, model_name="gpt-4o", openai_api_key="<<Open AI
API Key>>")

# Create a prompt template for multiple predictions
prompt_template = PromptTemplate(
    input_variables=["prompt"],
    template="{prompt}"
)
# Chain to generate multiple predictions
llm_chain = LLMChain(llm=llm, prompt=prompt_template) #A

# Function to simulate predictions with confidence scores
def get_prediction_with_confidence(prompt):
    # Simulating multiple predictions
    response = llm_chain.run(prompt)
    # Simulate confidence score (in a real-world case, you might use an API that
    # returns a confidence score)
    confidence_score = random.uniform(0.7, 0.99) # Simulating a confidence score
    return response, confidence_score #B

# Step 4: Make multiple predictions
num_predictions = 3 # Let's simulate 3 different predictions
predictions = []
for _ in range(num_predictions):
    prediction, confidence = get_prediction_with_confidence(enriched_prompt)
    predictions.append({"prediction": prediction, "confidence": confidence})
#C

# Step 5: Select the prediction with the highest confidence
best_prediction = max(predictions, key=lambda x: x["confidence"])
# Output the best prediction
print(f"Best Prediction:\n{best_prediction['prediction']}")
print(f"Confidence Score: {best_prediction['confidence']:.2f}") #D
```

#A Initialize the OpenAI API, create a prompt template and the LLMChain object.

#B Create a function to return a confidence score for the response generated from an input prompt.

```
#C Loop through a configurable number of prediction count and get the prediction with the confidence.
#D Select the best response based on the confidence score and print it.
```

Listing 5.5 shows the part we did not cover in the manual prompts mainly because manually generating multiple responses and selecting one is tedious and better suited for programming. We initialize the OpenAI API, use the prompt template to create the prompt based on Listing 5.4 and create an LLMChain object. We run the prompts for three iterations and select the response with the highest confidence. If you are interested in doing so, you could run this code for more than three iterations and see how the model responds.

MODEL HALLUCINATION

Models, even with extensive training, hallucinate and may end up generating incorrect responses. We should include a validation step after generating each response to mitigate this. You could do this by connecting to a reliable validation data set and then comparing the response from the model against the validated data set. You can discard the hallucinated responses and keep the accurate ones. You can also reduce this hallucination by fine-tuning the model for your specific use case. In this case, you can fine-tune the model with the validated data set. This will reduce the hallucination as the model will rely on the tuning data set for responses.

Now that we have learned about Generated knowledge prompting, we will examine their use cases and their limitations and mitigations.

USE CASES

Generated knowledge prompting is a technique in which a language model uses its generated knowledge to address complex problems. This approach is essential when fundamental knowledge is necessary to connect the different parts of the question and then generate a response for the prompt. Here are a few prominent use cases where generated knowledge prompting can be highly effective.

Scientific Research and Literature Reviews: Imagine you are analyzing medical records, searching for relevant information from clinical research, or diagnosing a patient based on their symptoms. In these situations, finding the correct insights is crucial. This is where Generated Knowledge Prompting can assist in retrieving and structuring information from extensive medical databases and research papers. However, it is important to note that standard language models have a fixed training cut-off date and do not automatically update their knowledge. For instance, a model trained on data up to 2023 will not have insights from studies published afterward. To address this, various techniques can be used to keep model interactions current:

1. **Retrieval-Augmented Generation (RAG):** This technique allows the model to retrieve real-time data from external sources, such as medical databases or research repositories, ensuring that responses reflect the most recent information.

2. **Reinforcement Learning from Human Feedback (RLHF):** This approach helps improve model accuracy and alignment with human expertise over time by incorporating structured feedback loops from domain experts.
3. **Continual Fine-Tuning:** By periodically updating the model with newly available datasets, fine-tuning allows it to adapt to evolving knowledge and refine its responses in specific domains like healthcare, finance, or legal analysis.

Software Development (Code Debugging and Optimization): When working on large-scale product development, it's essential to reuse existing components or understand complex software architectures. You can quickly access relevant code snippets, best practices, and algorithmic solutions based on extensive programming knowledge and documentation. The greatest advantage is that the model suggests code and explains its reasoning, helping you understand why a specific solution is effective. This approach is far more powerful than relying solely on static documentation or forums, as the model dynamically generates context-aware insights tailored to your specific coding challenges.

We have talked about a few use cases. However, can you think of more use cases where this prompting technique can be applied? Let's now look at limitations of this technique and possible mitigations.

LIMITATIONS & MITIGATIONS

Generated Knowledge Prompting is powerful and improves the response with static knowledge retrieval, but it has limitations. One major limitation is the potential for the model to generate inaccurate or outdated information, especially in fields where data changes rapidly, like healthcare and law. To mitigate this, updating the databases and external repositories is essential. Another limitation is that the model may not fully understand highly technical or niche queries, resulting in less relevant or useful responses. To address this, fine-tuning the model for specific use cases, breaking down complex problems into simpler, more specific questions can help the model provide clearer and more accurate answers. Additionally, incorporating human or computational oversight to validate the model's output can reduce the risk of relying on faulty or incomplete information.

5.6 Graph of Thought Prompting

The Graph-of-Thought (GoT) prompting technique enhances models' reasoning capabilities and helps them solve complex problems. Unlike Chain-of-Thought and Tree-of-Thought, which rely on linear or branching reasoning paths, GoT structures problem-solving as a network of interconnected reasoning steps. This approach aligns closely with multi-agent prompting, where different agents (or reasoning nodes) specialize in specific sub-problems and interact to refine solutions collectively. They rely on linear and branching techniques to solve sub-problems to solve the larger problem. However, the processing is linear even with the branching technique employed in Tree-of-Thought prompting. This means the model parses one branch to its node and generates a probable solution. The model does this for many repetitions until it converges onto the best-suited solution for the problem. However, this is not always the case in real-world scenarios. Many problems require information to be gathered and analyzed in connection with each other to come to conclusions. This is where graph-of-thought prompting comes into use. As the name suggests, the prompt is represented in terms of nodes and vertices, where each node represents a critical area of thought, often acting as an independent reasoning agent specializing in a particular aspect of the problem. These agents collaborate by exchanging information, refining hypotheses, and ensuring consistency across reasoning paths. The edges between nodes represent their interdependencies, allowing dynamic updates as new insights emerge.

Below, we see a way to guide the model. While we instruct the model to use the Graph-of-Thought (GoT) prompting technique, it is essential to note that models do not inherently understand this concept. Simply stating 'use GoT' is not enough—models require structured prompts that explicitly define the relationships between thoughts, hypotheses, and dependencies. In the following example, we structure the prompt to guide the model in simulating a reasoning graph.

Use the graph of thought prompting technique for this problem. Problem: A robbery has happened at the First Republic bank in Chicago. How do we start the investigation. Assume that you have all resources such as cameras, CCTV, license plates of all cars in the vicinity for up to two hours before and after the robbery.

Response

The Graph of Thought Prompting technique is a structured approach to solving problems by breaking them down into key nodes and connections. Each node represents a critical area of thought, and the connections represent logical relationships or dependencies between them. For this robbery investigation, let's construct the graph step-by-step.

Central Node: ****Robbery at First Republic Bank****

- This is the main event, and the investigation aims to solve this case by uncovering suspects, motives, and gathering evidence.

Key Nodes:

1. ****Crime Scene Analysis****

- ****Subnodes****:

- ****Physical evidence****: Fingerprints, footprints, weapon traces.

- ****Witnesses****: Statements from bank employees, customers.

- ****Exit/Entry Points****: Any signs of forced entry or tampering.

- ****Next Steps****:

- Examine the crime scene for physical clues.

- Interview witnesses for initial details about the robbers (e.g., appearance, behavior).

- Assess how the robbers entered and exited the bank.

This example shows how the model tries to solve the problem using the graph-of-thought prompting technique. Although the response is big, it contains key elements defining this prompting technique. The model's problem is to figure out who robbed the bank. We provide the model with some data sources and then ask it to generate a solution framework to solve the problem. The model does this by first defining key nodes. Each of the nodes is then divided into sub-nodes. The model then gathers relevant information from these sub-nodes and forms connections between nodes. Let's see how the model first creates a central node.

Response

Central Node: Robbery at First Republic Bank

This is the main event, and the investigation aims to solve this case by uncovering suspects, motives, and gathering evidence.

Key Nodes:

1. Crime Scene Analysis

- Subnodes:
 - Physical evidence: Fingerprints, footprints, weapon traces.
 - Witnesses: Statements from bank employees, customers.
 - Exit/Entry Points: Any signs of forced entry or tampering.
- Next Steps:
 - Examine the crime scene for physical clues.
 - Interview witnesses for initial details about the robbers (e.g., appearance, behavior).
 - Assess how the robbers entered and exited the bank.

As shown in the response above, the model creates the central node based on the context provided in the prompt. Next, the model creates key nodes that will be required to solve the case. The first key node is the crime scene analysis, under which the model, based on its training data, shows what needs to be done. These steps form the sub-nodes of the key node "Crime Scene Analysis." The sub-nodes consist of gathering physical evidence and witness statements and assessing the behavior of the robbers by analyzing the points of entry and exit. For each of the sub-nodes, the model also defines the next steps. In this case, the next steps are to gather evidence, etc. Isn't this breakdown intuitive? Imagine multiple such cases, and the model is provided with contextual information and data sources. An auto-assigning system could assign each sub-node to case officers based on their workloads. The workloads could be taken into consideration when assigning the sub-node to officers. Let's see a few more key nodes generated by the model.



2. CCTV Footage Analysis

- Subnodes:
 - Bank's Cameras: Focus on all cameras inside and outside the bank.

- Nearby Street Cameras: Collect footage from nearby traffic or public street cameras.
- Time Range: Check footage 2 hours before and after the robbery to track suspicious activities.
- Next Steps:
 - Identify suspicious individuals and vehicles.
 - Trace movements of suspects leading up to and after the robbery.
 - Use facial recognition if possible to identify known criminals.

3. License Plate Tracking

- Subnodes:
 - List of Plates: Collect license plates in the vicinity during the given timeframe.
 - Cross-reference: Compare license plates with known offenders.
 - Out-of-place vehicles: Look for any vehicles not typically present in the area.
- Next Steps:
 - Use traffic databases or DMV records to identify vehicle owners.
 - Check for stolen vehicles or fraudulent plates.

As shown in the response above, each of these nodes relates to the information provided in the prompt. In the prompt, we stated that information from CCTV cameras, license plates, etc., is available for analysis. The model considers this to create nodes and uses the nodes to create sub-nodes and the next steps for each sub-node. We see the model's ability to disambiguate an ambiguous problem as it creates additional nodes.



4. Suspect Profiles

- Subnodes:
 - Known Offenders: Check local databases for robbers with a history of bank crimes.
 - Behavioral Patterns: Assess the sophistication of the robbery (e.g., amateur vs. professional).
- Next Steps:

- Cross-reference any suspect profiles with CCTV and witness accounts.
- Create a list of potential suspects.

5. Escape Routes & Timing

- Subnodes:
 - Exit Strategy: Investigate how the robbers left the scene—car, foot, or otherwise.
 - Roads and Traffic Patterns: Look at traffic patterns to identify possible escape routes.
 - Distance from Scene: How far could suspects have traveled in the time since the robbery?
- Next Steps:
 - Map out all potential escape routes.
 - Check CCTV along those routes for further clues.

6. Internal Connections

- Subnodes:
 - Bank Employees: Investigate whether there was inside help or prior knowledge.
 - Alarm Systems: Check if the alarm was triggered correctly; if not, look for tampering or disabling.
- Next Steps:
 - Interview bank employees to identify any unusual behavior.
 - Check for security system anomalies or hacks.

7. Forensics

- Subnodes:
 - DNA Evidence: Gather DNA from any traces left behind (e.g., hair, blood).
 - Weapon/Tool Analysis: Analyze weapons or tools used to breach the bank's security.
- Next Steps:
 - Run forensic analysis on all evidence.
 - Check databases for DNA or other biological matches.

Note that the key nodes, sub-nodes, and the next steps generated above are not defined or provided in the context. The model does so based on its training dataset and the ability to extend the problem statement's possible solutions beyond what is defined in the prompt. The model extrapolates and forms nodes that discuss creating suspect profiles, analyzing escape routes and timings, and internal connections. In the next step, the model generates connections between all the key nodes. Next, let's look at how the model connects the nodes and sub-nodes.

Connections Between Nodes:

- CCTV Footage connects with License Plate Tracking and Suspect Profiles because footage may reveal key suspects and their vehicles.
- Escape Routes connects with License Plate Tracking as it helps identify vehicles leaving the scene.
- Crime Scene Analysis connects with Forensics and Internal Connections to determine how the robbery was carried out and whether there was inside help.
- Witnesses connects with Suspect Profiles to narrow down the list of suspects based on descriptions.

The model then generates hypotheses. These hypotheses consider the key nodes, sub-nodes, and the connections between each node. Below shows the hypotheses generated by the model.



Hypotheses and Next Steps:

- Hypothesis 1: The robbers used a stolen vehicle to escape.
- Focus on cross-referencing license plates with stolen vehicle reports.
- Hypothesis 2: The robbers had inside help.
- Focus on interviewing employees and checking their backgrounds for possible connections to the robbers.
- Hypothesis 3: The robbers are professionals and may have targeted similar banks before.
- Focus on matching the modus operandi with previous robberies.

By systematically analyzing each node and its connections, the investigation can proceed logically from gathering basic evidence to constructing profiles and eventually narrowing down suspects.

Imagine you could programmatically prompt the model to solve this problem. You could add real data sources that allow you to play out each key node and sub-node by assigning each task to case officers or other AI agents to sift through data and present it to a human case officer to validate and verify. You could allow the AI to ingest all the data and generate a possible suspect list or point to a list of suspects with probabilities attached. Let's try this example programmatically with LangChain. While the UI example provides a simple prompt to the model and relies on its internal reasoning capabilities, the LangChain implementation incorporates additional structured information. This added context improves accuracy and consistency, particularly in scenarios requiring step-by-step reasoning or multi-step problem-solving. Listing 5.6 shows the same problem being presented as a Python program. Since the program will encompass the entire problem, including the creation of key nodes and sub-nodes, we will analyze this program in parts.

Listing 5.6 shows a Python and LangChain implementation of the problem.

```
from langchain import LLMChain, PromptTemplate
from langchain.llms import OpenAI

# Step 1: Define the prompts for each node in the investigation

# Crime Scene Analysis Node
crime_scene_template = """
You are an investigator analyzing a robbery at the First Republic Bank. Given the
crime scene, evaluate the following:
- What physical evidence can be gathered?
- Are there any witness statements?
- How did the robbers enter and exit the bank?

Respond with a list of potential evidence and actions to take.
"""

crime_scene_prompt = PromptTemplate(input_variables=[],
template=crime_scene_template) #A

# CCTV Footage Analysis Node
cctv_template = """
You have access to CCTV footage from both the bank and nearby areas. Focus on the
following:
- Check for suspicious individuals entering or leaving the bank two hours before
and after the robbery.
- Identify any vehicles used for the getaway.
- Use facial recognition if possible.

What do you find? Respond with key observations and suspects.
"""

cctv_prompt = PromptTemplate(input_variables=[], template=cctv_template) #B

#A Create a crime scene prompt using the prompt text and prompt template.
#B Create a CCTV prompt using the prompt text and prompt template.
```

Listing 5.6 defines the prompts for every node in the program and Listing 5.7 shows the remaining prompts we created using a prompt template.

Listing 5.7 shows the remaining prompt templates.

```
# License Plate Analysis Node
license_plate_template = """
You have a list of license plates from vehicles in the vicinity of the bank two
hours before and after the robbery.
- Cross-reference these plates with known offenders.
- Identify any stolen or out-of-place vehicles.

Which vehicles seem suspicious? Respond with details of vehicles and their
owners.
"""

license_plate_prompt = PromptTemplate(input_variables=[],
template=license_plate_template) #A

# Suspect Profiles Node
suspect_profile_template = """
You are building a profile of potential suspects based on the robbery.
- Cross-reference with known offenders in the database.
- Use witness descriptions and CCTV footage to match with existing criminal
profiles.

What is the profile of the most likely suspect(s)?
"""

suspect_profile_prompt = PromptTemplate(input_variables=[],
template=suspect_profile_template) #B
```

#A Create the license plate analysis node using a prompt template.

#B Create the suspect profile node using a prompt template.

Next, we define the execution chain for each of these nodes and then call each chain. Listing 5.8 shows the creation of model chains and how they are called for execution.

Listing 5.8 shows the model chains and their execution.

```
# Step 2: Define the chains

llm = OpenAI() # You can replace this with your LLM provider

# Crime Scene Chain
crime_scene_chain = LLMChain(llm=llm, prompt=crime_scene_prompt) #A

# CCTV Footage Chain
```

```

cctv_chain = LLMChain(llm=llm, prompt=cctv_prompt). #B

# License Plate Chain
license_plate_chain = LLMChain(llm=llm, prompt=license_plate_prompt) #C

# Suspect Profiles Chain
suspect_profile_chain = LLMChain(llm=llm, prompt=suspect_profile_prompt) #D

# Step 3: Chain the nodes together logically (Graph of Thought Structure)

def solve_bank_robbery():
    # Crime Scene Analysis
    crime_scene_output = crime_scene_chain.run()
    print(f"Crime Scene Analysis: {crime_scene_output}")

    # CCTV Footage Analysis
    cctv_output = cctv_chain.run()
    print(f"CCTV Analysis: {cctv_output}")

    # License Plate Analysis
    license_plate_output = license_plate_chain.run()
    print(f"License Plate Analysis: {license_plate_output}")

    # Suspect Profile Generation
    suspect_profile_output = suspect_profile_chain.run()
    print(f"Suspect Profile: {suspect_profile_output}")

# Step 4: Execute the chain
solve_bank_robbery() #E

```

#A Create a crime scene chain.

#B Create a CCTV chain.

#C Create a license plate chain.

#D Create a suspect profile chain.

#E Call the solve_bank_robbery function that calls each of the chains created earlier.

We define each chain with the nodes created earlier and then call each chain to run via the model. The model generates an output at each step. In the program, we show how each step can be run. However, we do not do anything with the outputs. Can you think of the actions that we could perform with the outputs? You are correct if you think about verification, validation, and hypotheses runs. After each output from the chain, you could run validations, use the output to modify the next nodes, and then run the chain again. Finally, you can generate hypotheses and then evaluate pairs of hypotheses until you have the most viable solution. Alternatively, in real-world scenarios, you could surface the output at each stage to the user for human validation before it is used in the next node or sub-node. Let's look at this prompting technique's use cases, limitations, and mitigations.

USE CASES

Multi-Agent Simulation: GoT closely resembles multi-agent prompting, where different agents independently reason about specific parts of a problem while exchanging insights. Each agent (or node in the graph) can represent an expert in a given subtask, such as analyzing CCTV footage, forensic evidence, or suspect profiles. This decentralized approach improves accuracy, reduces bias, and enhances reasoning efficiency. Taking the same example we saw earlier about the robbery, we could have multiple agents simulated, each responsible for solving one node and its corresponding sub-nodes while interacting with the other agents for information sharing.

Game Design & Strategy: We all have played video games while growing up. Graph of thought prompting can enable NPC behavior within these video games to be more realistic and dynamic. Depending on player choices, the NPC behavior could change. Behind the scenes, a decision tree powered by the graph of thought prompting technique can enable dynamic, non-linear, and thoughtful responses from NPCs and other characters in the game.

Education: Learning is unique to every individual. Over the years, many learning techniques have been used across schools and universities. With Gen AI, education and learning can improve by incorporating graph prompting. Graph prompting can help map the relationship between learning objectives and concepts. With graph prompting, educators, researchers, and entrepreneurs can create personalized learning pathways that suggest how students with different learning abilities can progress through complex subjects.

We encourage you to have a think about more use cases where graph of thought prompting can be used. Let us now look at some of the limitations of this prompting technique and ways to mitigate them.

LIMITATIONS & MITIGATIONS

While this prompting technique is much better at generating reasoning chains and possible solutions than Chain-of-thought and Tree-of-thought prompting techniques, it does have a few limitations. Most of the limitations stem from this complex prompting technique requiring a skilled prompt engineer or tools that can help you craft effective prompts and require models to interpret the technique. Let us discuss each of the limitations and their mitigations.

Implementation Complexity: As we have seen in the examples starting in Listing 5.4, the construction of this prompting technique is complex. If you are starting from scratch for a similar problem, as discussed in the example, you need to craft these complex prompting techniques. In some cases, you could also end up with rigid prompts due to the structure of the central node, key nodes, and sub-nodes. So, how do we get past this limitation? The easiest way is to follow what we learned in the advanced chain-of-thought prompting technique. You can ask the model to generate the central node, key nodes, sub-nodes, next steps, and hypotheses prompts based on the problem statement. We saw this in Listing 5.4.

Model Limitations: Many models do not understand the graph structure. They can follow instructions that resemble a graph, but they inherently do not know how a node relates to the sub-nodes, etc. Therefore, choosing the model or the ability to guide the model to solve the problem resembling a graph matters a lot.

Scalability: The larger the problem, the larger the graph, and the more challenging it will be for the model to parse the graph and compute the result. You could also run into prompt length limits. A graph-based problem will have prompts for the central node, key nodes, sub-nodes, and all other steps. To mitigate this, you can divide the problem into smaller parts, limiting the graph and the prompt lengths. This could solve the issue of scaling.

Interpretation Challenges: Graph-of-thought prompting requires complex prompts for the central, key, and sub-nodes. The problem here is the prompt's effectiveness. How effective is the prompt, and is it easy enough for the model to comprehend? To mitigate this, you could use the model to generate prompts for each node and define the connections clearly to ensure that the model can understand the connection between each node. Using examples to guide the model in solving the problems will also be beneficial.

5.7 Multi-modal Prompting

As the name suggests, multi-modal prompting uses multiple input types to guide the model in generating responses. The input types can be text, video, image, or audio. The varied inputs can help the model understand the context better and improve the output generated. You could use all modes of input simultaneously to prompt the model. However, it should be noted that this technique works only with models that support multi-modal inputs.

How does it work? It works much like all other prompting techniques we have discussed. The model is trained on multiple datasets, including text, video, audio, images, etc. During its training, the model maps the relationships between each input data and different modalities of inputs. When a user inputs data or creates a prompt combining two or more (image, video, text, or audio) of the input types, the model parses the inputs and builds a comprehensive understanding by cross-referencing the inputs.

For example, if you prompt a model with the image of a mountain and ask it to describe or identify it, then the model first parses the image and then your prompt to understand the question. Once understood, based on its training data, the model tries to answer your question. There are, however, a few types of multi-modal prompting techniques. Let's look at a few important ones.

Concatenated Multi-Modal Prompting

As the name suggests, this technique involves concatenating multiple modalities into a single input. The model processes this input and generates a response. For example, you might want the model to define a painting and reflect on its colors. In this case, you would prompt the model with an image of the painting and a text prompt asking the model to analyze it. The model would combine the image and the prompt and generate a response.

Sequential Multi-Modal Prompting

In this technique, the prompts of different modalities are parsed one after the other. For example, you might want the model to parse an image first, then based on the analysis; you want to ask more questions or refine the image. This could be an iterative process until you get the image you want.

Cross-Attention Multi-Modal Prompting

Cross-Attention Multi-Modal Prompting allows the model to understand relationships between different types of input by paying attention to how they interact. Instead of processing text and images separately, the model aligns their meanings to generate a richer response. For example, you want the model to caption a video or an image. In this case, the model parses the image and generates captions. Conversely, you could use this technique to parse an image or a video and then have the model answer questions based on the parsed data.

Transformative Multi-Modal Prompting

Transformative Multi-Modal Prompting converts one type of input into another to improve interaction. The model first transforms the input (e.g., speech-to-text) before processing it.

Do you want to try out this technique using a model? This is one of the more fun prompting techniques to work with. We will not go into examples for this prompting technique as presenting a rich example using text is not feasible. You could use a combination of modalities such as image, text, and audio to see how this prompting technique works.

Let's examine some of this prompting technique's most important use cases.

USE CASES

Content Generation/Modification: Multi-modal prompts combining text with images or audio can generate or modify content, such as creating visual effects for movies or producing custom illustrations based on textual prompts. This technique can also be used for real-time content generation, such as video translation, captioning, etc.

Accessibility Tools: Multimodal AI significantly enhances accessibility tools, such as assistive technologies that convert visual data to audio descriptions for visually impaired users. Speech-to-image tools can create visual aids from spoken prompts, supporting those with hearing impairments or learning disabilities. Multimodal search engines allow users to refine their queries using text and images, facilitating more precise search results. These systems are valuable for extracting data from scanned documents by integrating text and image analysis.

Augmented and Virtual Reality: Augmented reality (AR) and virtual reality (VR) benefit from multimodal interactions that merge audio commands with visual overlays for richer user experiences. Combining text scenarios with audiovisual data can make training simulations more realistic. In smart home and IoT integration, voice-controlled devices with visual feedback provide intuitive interactions, and monitoring systems analyze audio and images for context-aware security alerts.

What other use cases can you think of? Can this technique be used for customer support when returning defective items?

LIMITATIONS & MITIGATIONS

Even with the diverse use cases, multi-modal prompting techniques have some limitations. Let's look at the limitations and their mitigations.

Data Integration Complexity: Integrating and aligning different modalities (e.g., text, images, audio) can be challenging due to the diverse nature of data structures. Processing each type requires specialized techniques and combining them while maintaining coherence adds complexity to the model design. This can be mitigated using advanced data fusion techniques, such as attention mechanisms and cross-modal transformers, which can help align and combine data from different modalities more effectively. Additionally, breaking down the integration process into stages (early, middle, or late fusion) and selecting the appropriate approach based on the use case can improve results.

Computational Costs: Training multi-modal models requires extensive datasets and significant computational power. Processing diverse data types together is more resource-intensive than handling a single modality, resulting in longer training times and higher energy consumption. Leveraging pre-trained models and transfer learning can reduce the amount of data and computation needed for training. Fine-tuning models on specific tasks instead of training from scratch helps save time and resources. Utilizing cloud-based solutions and distributed computing can also manage computational demands more efficiently.

Scalability: Scaling multi-modal systems to handle high volumes of data across different modalities, especially in real-time scenarios, can be challenging. The increased complexity can impact system performance and user experience. To mitigate and make this technique scalable, you can use distributed and scalable architectures, such as microservices that handle different modalities separately and combine results, enhancing scalability. Implementing load balancing and resource management strategies also helps distribute the computational load efficiently.

5.8 Automatic Prompt Engineering

Automatic Prompt Engineering aims to reduce the need for human expertise in designing these prompts by using algorithms and optimization techniques. The language model automates the discovery of optimal prompts, reducing the dependency on manual trial and error methods. Automatic Prompt Engineering leverages machine learning methods, especially reinforcement learning and neural architectures, to autonomously generate, validate, select, and refine prompts. You can incorporate predefined evaluation metrics such as accuracy, clarity, conciseness, relevance, or any other metrics suitable to your use case. Figure 5.5 shows a diagrammatic representation of how automatic prompt engineering works.

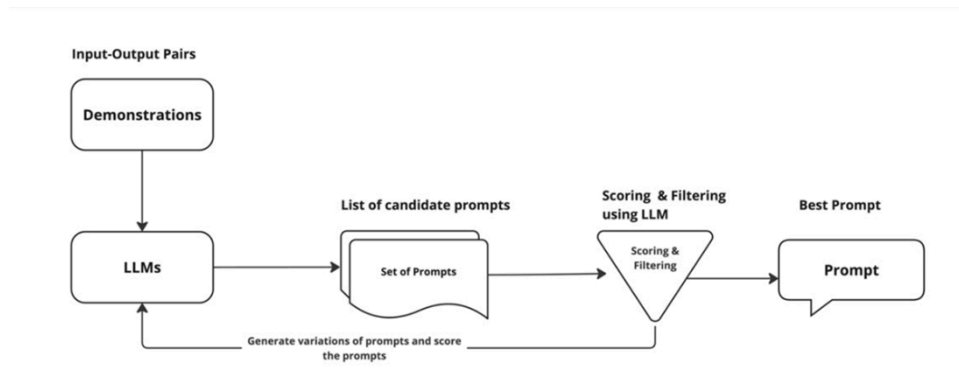


Figure 5.5 shows how Automatic Prompt Engineering works.

Figure 5.5 shows how Automatic Prompt Engineering (APE) works. The process is like working with a new search engine to see how it parses your search queries to fetch results. In this case, we feed the model several types or variations of prompts and allow it to generate a response to each prompt. Then, responses are evaluated based on several criteria, such as clarity, accuracy, conciseness, and relevance. We further generate variations of those prompts and keep scoring them. Finally, the best prompt is selected based on clarity. In essence, we leverage the model to engineer effective prompts and guide the model to generate accurate and relevant responses. Imagine you want to generate an optimized prompt for summarizing a technical article on quantum computing. There are two ways to approach this: (1) summarizing a specific article by referencing its content or (2) generating a general summary based on the model's existing knowledge. In this example, the model works without a specific source article and summarizes based on its pre-trained knowledge. If we wanted the model to summarize a specific article, we would need to provide the full text or key passages as part of the prompt.

Generate Initial Prompt Variations: Generate and manually draft a variety of prompts on quantum computing to produce different types of summaries, ranging from technical to simplified language. Here are several prompt variations:

- Prompt 1: Summarize Quantum Computing in simple terms.

- **Prompt 2:** Can you explain the key points about Quantum Computing for beginners?
- **Prompt 3:** What are the main ideas behind Quantum Computing? Can you summarize it in an easy-to-understand language?
- **Prompt 4:** Provide a simplified overview of Quantum Computing, highlighting the essential points.
- **Prompt 5:** Write a summary of Quantum Computing suitable for someone without a technical background.

Each prompt is then fed as input to the model individually, and the response generated is as follows.

Response to Prompt 1

Quantum computing is a new type of computing that leverages the principles of quantum mechanics, a branch of physics that explains how very tiny particles like atoms and electrons behave. Here's a simple way to understand it:

In traditional computers, information is stored in bits, which can be either a 0 or a 1. Quantum computers, however, use **quantum bits** or **qubits**. Unlike bits, qubits can be 0, 1, or both at the same time, thanks to a concept called **superposition**. This allows quantum computers to process multiple possibilities at once, making certain calculations much faster than classical computers.

Another concept is **entanglement**, where qubits become linked, so the state of one qubit can depend on the state of another, even across long distances. This can lead to highly coordinated computations.

Quantum computing holds promise for solving complex problems that are too hard for classical computers, like simulating molecules for drug discovery or optimizing large systems. However, it's still an emerging field, with challenges around maintaining qubit stability and building scalable machines.

Response to Prompt 2

Sure! Here's a beginner-friendly breakdown of the key points in quantum computing:

1. **Quantum Bits (Qubits)**

- **What they are:** In classical computers, the basic unit of data is a bit, which can be either 0 or 1. Quantum computers use qubits, which can be 0, 1, or both at the same time due to **superposition**.

- **Why they matter**: This ability to hold multiple states at once gives quantum computers an edge in solving complex problems.

2. **Superposition**

- **What it means**: Superposition allows qubits to exist in multiple states simultaneously.
- **Why it's useful**: This lets quantum computers explore many possibilities at once, making certain calculations faster.

3. **Entanglement**

- **What it is**: Entanglement is a connection between two qubits where the state of one qubit is directly related to the state of another, no matter the distance between them.
- **Why it's powerful**: Entangled qubits work together in a coordinated way, enabling faster and more complex computations.

4. **Quantum Interference**

- **What it does**: Quantum interference controls the likelihood of a qubit's possible outcomes, helping quantum algorithms to find the right answer by amplifying correct paths and canceling out incorrect ones.

5. **Applications**

- **Potential uses**: Quantum computing is promising in fields like cryptography, medicine, artificial intelligence, and materials science.
- **Why it's valuable**: Problems that would take classical computers years to solve could potentially be solved by quantum computers in seconds.

6. **Challenges**

- **Stability**: Qubits are highly sensitive to their environment, so they can lose their state quickly (a problem called **decoherence**).
- **Scalability**: Building large, stable quantum computers with many qubits is very challenging.

In short, quantum computing is all about using the strange rules of quantum mechanics to process information in ways that classical computers can't, opening doors to solving very complex problems.

You could try running the other prompts to and check the response generated by the model. Feel free to add more variations of the prompt we used or use a different example altogether.

Evaluate Responses Based on Predefined Criteria: Evaluate the responses based on several criteria, including clarity, accuracy, conciseness, and relevance. This task is complex, so the evaluation metrics will be more nuanced than correctness. Let's create a prompt that outlines the criteria for assessing the quality of the responses.

Score each response using a simple scale (e.g., 1 to 5) based on these criteria and include an overall score.

Clarity: How easy is the response for a non-expert to understand?

Accuracy: Does the response accurately reflect the main ideas of quantum computing?

Conciseness: Is the response concise without omitting essential details?

Relevance: Does the response focus on the critical points of the article?

Response:

Response to Prompt 1:

Clarity:4, Accuracy:5, Conciseness:4, Relevance:4

Overall Score:17/20

Response to Prompt 2:

Clarity:5, Accuracy:4, Conciseness:5, Relevance:4

Overall Score:18/20

Response to Prompt 3:

Clarity:4, Accuracy:5, Conciseness:4, Relevance:4

Overall Score:17/20

Response to Prompt 4:

Clarity:3, Accuracy:5, Conciseness:4, Relevance:5

Overall Score:17/20

Response to Prompt 5:

Clarity:4, Accuracy:5, Conciseness:5, Relevance:4

Overall Score:18/20

Refine and Score Prompts: Choose the best prompt based on its evaluation score. If two or more prompts receive the same score, select the one that prioritizes clarity while staying relevant and concise. This is especially important when explaining complex topics to non-experts. In this example, Prompt 2 is the best choice because it has the highest score for clarity.

Select the Best Prompt: In an automated process using programming, this technique can continuously improve through a feedback loop. If users provide feedback to simplify the response, the language model might generate a new set of prompts. For example:

- Explain Quantum Computing in simpler terms.
- Summarize Quantum Computing with a focus on real-world applications.

Now that we have seen how you can manually use this approach for prompt engineering, we can look at how to accomplish the same process using programming. As with past examples, we will use LangChain to code the Automatic Prompt Engineering process. Listing 5.9 shows the programmatic implementation of APE.

Listing 5.9 shows prompt templates, evaluation criteria, and ideal words for the prompt responses.

```
from langchain import PromptTemplate, LLMChain
from langchain_openai import ChatOpenAI
import os

# Define different prompt templates
prompt_templates = [
    "Summarize Quantum Computing in simple terms.",
    "Can you explain the key points about Quantum Computing for beginners?"
    "What are the main ideas behind Quantum Computing? Can you summarize it in an
    easy-to-understand language?",
    "Provide a simplified overview of Quantum Computing, highlighting the essential
    points.",
    "Write a summary of Quantum Computing suitable for someone without a technical
    background."
]

# setting the openai API key as environment variable
os.environ['OPENAI_API_KEY'] = "<<YOUR_OPENAI>>" # replace with your openai API
key

llm = ChatOpenAI(temperature=0, model_name="gpt-4o",
openai_api_key=os.environ['OPENAI_API_KEY'])

# Evaluation criteria weights (these can be customized)
criteria_weights = {
    'clarity': 1.0,
    'accuracy': 1.5,
    'conciseness': 1.0,
    'relevance': 1.5
}

# Example of an expected "ideal" summary (used for accuracy/relevance evaluation)
ideal_summary_keywords = ["quantum", "computing", "qubits", "superposition",
"entanglement", "faster"]
```

Now that we have learned about the Automatic Prompt Engineering technique, we will examine its use cases, limitations, and mitigations.

USE CASES

Automatic Prompt Engineering is a technique to minimize human involvement in prompt creation, making it easier and faster to extract relevant information using language models. This technique has various applications across various industries and domains, mainly where language models are used. Below are some notable use cases:

Healthcare AI: Imagine you are a healthcare provider needing to summarize a patient's medical history, retrieve relevant clinical guidelines, or generate treatment recommendations based on specific symptoms. In these scenarios, the accuracy and clarity of the information you gather are critical, often within a limited time frame. This is where Automatic Prompt Engineering (APE) can enhance the process by optimizing how queries are phrased to maximize the quality of LLM-generated responses. APE helps automatically generate and refine prompts to ensure the model retrieves the most relevant medical insights, personalized summaries, or diagnostic suggestions. This technique is particularly beneficial in complex cases where phrasing the question precisely can significantly impact the quality of the response. By automating the prompt generation, healthcare providers can save time, reduce cognitive load, and obtain clearer, more accurate results tailored to the needs of clinicians and patients, ensuring that the information delivered is precise and actionable.

Content Generation and Creative Writing: Consider you are tasked with writing a product description, crafting a blog post, or generating a creative story for a marketing campaign. In these situations, the content's tone, style, and structure are critical for engaging your audience. This is where Automatic Prompt Engineering (APE) can significantly enhance the creative process. APE automatically generates and optimizes prompts to guide LLM models in producing content that aligns with your goals, whether you need a playful tone, professional language, or a detailed narrative. By testing different prompts and refining them based on clarity, creativity, and relevance, APE allows the model to generate high-quality, tailored content quickly and efficiently. This approach saves time and helps ensure the output resonates with your target audience, offering a more dynamic alternative to traditional content creation methods.

Question Answering Systems: Imagine you're interacting with a system to answer customer questions, provide troubleshooting advice, or assist with academic inquiries. In these cases, the clarity and correctness of the answers are critical. Automatic Prompt Engineering (APE) makes question-answering smarter by automatically tweaking and optimizing how the questions are posed to the AI. This allows the system to produce responses that are not just relevant but also tailored to the user's needs. Whether the question is straightforward or complex, APE refines the prompts, making the AI better at retrieving the most accurate and focused answers. The beauty of this approach lies in its adaptability to adjust, continually improving the quality of responses in real-time. This results in a more interactive, personalized experience compared to rigid, pre-programmed answers, helping users get the information they need quickly and effectively.

LIMITATIONS & MITIGATIONS

Automatic Prompt Engineering (APE) is an effective technique that automates the generation and optimization of prompts to improve responses. However, it has its limitations. One major challenge is the complexity of navigating the extensive prompt space, which can result in inefficient searches and suboptimal prompt selections. To address this, employing search algorithms such as beam search or evolutionary strategies can help refine the options more effectively. Another limitation is that APE may have difficulty generalizing across different tasks, producing prompts that work well in one scenario but not in another. This issue can be mitigated by implementing domain-specific tuning or task-specific prompt templates.

Additionally, APE systems can inadvertently introduce or reinforce biases in the model's responses, especially if they optimize prompts based on biased datasets or evaluation metrics. Regular audits for bias and the incorporation of fairness criteria can reduce this risk. Lastly, APE often lacks human intuition, leading to technically correct but contextually irrelevant responses. Including human feedback in the evaluation process can help ensure that the generated prompts align with user expectations and the intended context.

We went through a lot of concepts in this chapter. By now, you should be able to leverage automatic and tree-of-thought prompting techniques to overcome issues of chain-of-thought prompting technique. To solve more complex problems, you should be able to leverage graph-of-thought prompting technique to explore interconnected nodes simultaneously and generate hypotheses culminating to the final solution. Extending from text-based prompts, you could leverage multi-modal prompting techniques to guide a model's response using different data types. Lastly, you can use Automatic Prompt Engineering as a feedforward technique to generate and refine prompts with validation steps, reducing human intervention. This helps when you want to solve problems at scale where the model's capability is tested to cater to a large set of use cases. We highly recommend you go through this chapter again if any of the concepts are unclear.

5.9 Summary

- We built upon the Chain of Thought prompting technique to discuss Automatic and Tree of Thought prompting. These techniques incorporate chain-of-thought prompting while mitigating some issues in regular chain-of-thought prompting techniques.
- Automatic chain-of-thought prompting is a technique where a model is guided to generate intermediate reasoning steps automatically before arriving at a final answer. This helps improve complex problem-solving by breaking down tasks into smaller, logical components, enhancing accuracy and interpretability.
- Tree of thought prompting is a technique where a model explores multiple reasoning paths or branches to solve a problem, allowing it to consider various solutions and their outcomes. This approach mimics human brainstorming, enhancing decision-making by evaluating different options before choosing the best response.

- Graph-of-Thought prompting extends the reasoning process by structuring problem-solving as a network of interconnected reasoning steps rather than a single linear path. This technique allows multiple reasoning paths to be explored simultaneously, making it particularly useful for complex decision-making, hypothesis generation, and investigative problem-solving.
- Generated knowledge prompting involves guiding a model to first generate relevant background information or context before addressing a main query. This technique helps the model build a foundation of knowledge, improving its ability to provide accurate and informed responses.
- Multi-modal prompting technique involves using inputs from multiple data types, such as text, images, and audio, to guide a model's response. This enables more complex and context-aware outputs by combining and understanding diverse sources of information.
- Automatic Prompt Engineering is a technique where models themselves generate and refine prompts to optimize task performance without human intervention. This approach helps in automating the process of finding effective prompts that enhance the model's output accuracy and efficiency.

6 Retrieval-Augmented Generation

This chapter covers

- Basics of Retrieval Augmented Generation (RAG): What is it, why is it used, and what benefits do we gain from RAG?
- A deep insight into how prompt techniques, elements, and patterns converge with RAG.
- Deep dive into the architecture and workflow of RAG and its components.
- Real-world examples leveraging the RAG workflow in its different phases.

Chapter 5 discussed Automatic and tree-of-thought prompting techniques. These techniques leverage chain-of-thought prompting while mitigating its limitations. We saw how graph-of-thought prompting can help solve problems that require interconnected reasoning. Furthermore, we saw how generated knowledge prompting can help leverage knowledge generation for smaller sub-problems while moving towards solving the main problem. Lastly, we saw the field of Automatic Prompt Engineering, which uses models to generate prompts, modify them based on feedback, and use prompts at scale.

In this Chapter, we will examine how Retrieval-Augmented Generation (RAG) plays an important role in interacting with models. We will discuss RAG's overall architecture, prompt engineering's role, retrieval methodologies' benefits and uses within RAG, the use cases where RAG is helpful, and the advantages and disadvantages of building a RAG-based system to interact with models. To start, let's understand what RAG is.

6.1 What is Retrieval-Augmented Generation

In traditional language models, responses are generated solely based on pre-learned patterns and information from the training phase. While this allows for fluent and coherent text generation, it also introduces significant limitations—models can hallucinate, provide outdated information, or fail to answer queries requiring specialized or real-time knowledge. Retrieval-Augmented Generation (RAG) overcomes these limitations by integrating a retrieval step into the response-generation process. Instead of relying purely on static training data, RAG dynamically retrieves relevant information from an external knowledge base before generating a response. Before we go further, let's look at some of the constraints of models and how they work today.

- **Hallucination:** Models still hallucinate and provide erroneous responses.
- **Outdated Knowledge:** Responses from the models are constrained by the model's training data.
- **Limited Domain Expertise:** Most models are generic in their field of expertise, making it challenging to leverage them for a specific domain.
- **High Tuning Costs:** If you want to invest in tuning a model, the costs can be high, depending on how much tuning is required.

RAG addresses these limitations of traditional language models by integrating a retrieval step into the response-generation process. This approach ensures that the responses are accurate and backed by relevant information, reduced hallucination, and the model can leverage on-demand readily available knowledge to respond to queries. Before we dive deeper, let's look at how a RAG system works.

PROMPT VS QUERY? In this chapter, we will use “prompt” and “query” interchangeably, as both terms convey the same meaning. A query is a user's prompt to obtain information in a RAG system.

6.2 How does a RAG system work?

A Retrieval-Augmented Generation (RAG) system operates in three phases:

- **Pre-Retrieval:** The system indexes the data and prepares it for easy retrieval during real-time operations.
- **Retrieval:** Upon receiving a query or prompt from a user or another system, the data is retrieved from the data collected in the pre-retrieval phase.
- **Post-Retrieval:** The data retrieved, and the user query are fed to the model. The model then generates responses based on prompt templates defined in the RAG system.

Let's examine each of these phases in more detail. In the later sections of this chapter, we will also explore the architecture and workflow of an RAG system.

6.2.1 Pre-Retrieval

This phase ensures that the system has structured, relevant, and contextual knowledge about the topic or sets of issues that you would like the RAG system to handle. The pre-processing phase is done, as the name suggests, in advance before the system is ready to handle any queries. Consider this to be synonymous with the tuning phase of a model. The big difference between the tuning of a model and the preprocessing phase is the cost. Preprocessing is often cheaper than tuning the model for your specific use case. The steps involved in the preprocessing phase are:

DATA COLLECTION

In this step, all data sources are gathered. Imagine you want to build a RAG system focused solely on cloud and distributed computing. In this context, the data sources may include research papers, data scraped from cloud providers, implementer blogs, and more. The data can be structured (databases), semi-structured (JSON, XML), or unstructured (PDFs, text files, scraped data from websites, blogs, etc.). During data collection, depending on the sources, you could either normalize the data into a single structured dataset or utilize a combination of structured, semi-structured, and unstructured sources working together to power the RAG system.

TEXT CHUNKING

This step pertains to unstructured information sources such as PDFs and text files. The idea of text chunking consists of dividing large documents into smaller sections to improve retrieval granularity. Chunking also aids in preserving context and ensures effective embedding-based similarity searches.

EMBEDDING GENERATION

Embeddings are dense vector representations of text, images, and other data types that encode semantic meaning in a continuous vector space. What does that mean? Simply put, embeddings are a way to convert words or sentences into numbers so computers can understand their meaning. For example, if a sentence contains the words "chair," "table," and "dog," then the words "chair" and "tables" are close to each other, whereas "dog" is farther away in the vector space. How does this help? Well, embeddings help AI systems to find relevant information based on meanings and not just exact words. How does this work? Once the chunking is complete, a pre-trained model converts each text chunk to a vector embedding. Several models are available, such as OpenAI's text embedding models. The embeddings capture the semantic meaning of the text.

INDEXING & STORAGE

The embeddings and associated metadata are stored in vector databases like FAISS, Pinecone, and Chroma DB. These databases enable efficient similarity-based searches when processing a user query.

WHAT ARE FAISS, PINECONE, CHROMA DB?

FAISS (Facebook AI Similarity Search) – An open-source library optimized for fast similarity search in large-scale datasets. It is widely used for high-speed approximate nearest neighbor (ANN) search, making it ideal for local or on-device retrieval.

Pinecone – A managed vector database designed for real-time, scalable, and cloud-based similarity search. It abstracts away infrastructure complexity, making it easy to deploy for applications that require dynamic retrieval.

Chroma DB – A lightweight, open-source embedding database tailored for storing and querying vector representations of text, making it useful for applications that integrate structured and unstructured retrieval.

6.2.2 Retrieval

This phase occurs when a user submits a query to the RAG system. In this phase, the RAG system goes through the following steps.

QUERY EMBEDDING

The user's query is transformed into an embedding using the same embedding model utilized during the pre-processing phase. This guarantees that the query and the indexed documents reside in the same vector space for similarity comparison.

RETRIEVAL FROM VECTOR DATABASE

In this step, the RAG system retrieves information based on a similarity search from the vector database. Based on similarity scores, the system retrieves the top K most relevant clusters, which are used to generate query responses.

CONTEXTUAL AUGMENTATION

The retrieved text chunks from the previous step are formatted and combined into a structured response. The system may apply additional reranking techniques to enhance the quality of the retrieved information, thereby generating a much better response.

6.2.3 Post-Retrieval

This phase occurs once relevant data is retrieved from the data sources. The system then sends the data and query to the model for processing.

GENERATION

In this step, the retrieved information is incorporated into the user's query and submitted to the model for processing. The model produces the final response, ensuring it aligns with the gathered knowledge. The user's query is typically expanded, and a predefined prompt template is utilized that also specifies the format of the model's response. Following is an example of a prompt template the RAG system can use. In the example below, there are placeholders for the retrieved context the user's query, followed by a template of how the response should look like. The RAG system uses the template below and, at every step, appends information into the template before the next step processes the query.

Answer the user's query only considering the context retrieved and appended below. Structure the response according to the format provided in the template.

Retrieved Context:

{Top-K retrieved chunks}

User Query:

{User's Question}

Structured Answer:

-Key Insights

-Explanation

-Citations

The template above is merely one example. Depending on how the RAG system is structured, you might have multiple templates that adhere to various prompt patterns.

In this section, we reviewed the high-level workings of an RAG system, including the phases and their steps. Let's now examine what role prompt engineering plays in a RAG system.

6.3 Prompt Engineering & RAG

So far, we have understood what RAG is and the high-level workflow. However, this leaves us with an important question: What role does prompt engineering play in RAG, if any? Furthermore, why are we discussing RAG in a book about crafting effective prompts? To understand this, let's consider a real-life example that we have likely encountered many times. When someone asks you a question on a specific topic, you respond to the best of your abilities. This means you answer the question based on the best information available to you at that moment. This is primarily how models operate today. Have you ever found yourself discussing a topic and recalling something closely related yet unable to connect it fully? Later, you may realize that you misremembered the article and need to correct yourself. Even though models today can collate information from the internet, they cannot efficiently correlate, validate, or leverage the information they find. Today, you can utilize RAG to guide the model in retrieving answers based on validated, verified, and correlated information. The RAG system internally employs prompt patterns and templates to query the model, retrieve responses, and format the responses according to the used template.

Let's clarify this with an example. In this scenario, we will use ChatGPT and utilize the following prompt on ChatGPT's web interface.

What was the average temperature in February 2024 in New York City?

Figure 6.1 shows the response from ChatGPT. You could programmatically or use the web interface to try the same prompt on other models. You can expect a similar response if the model can access the internet.

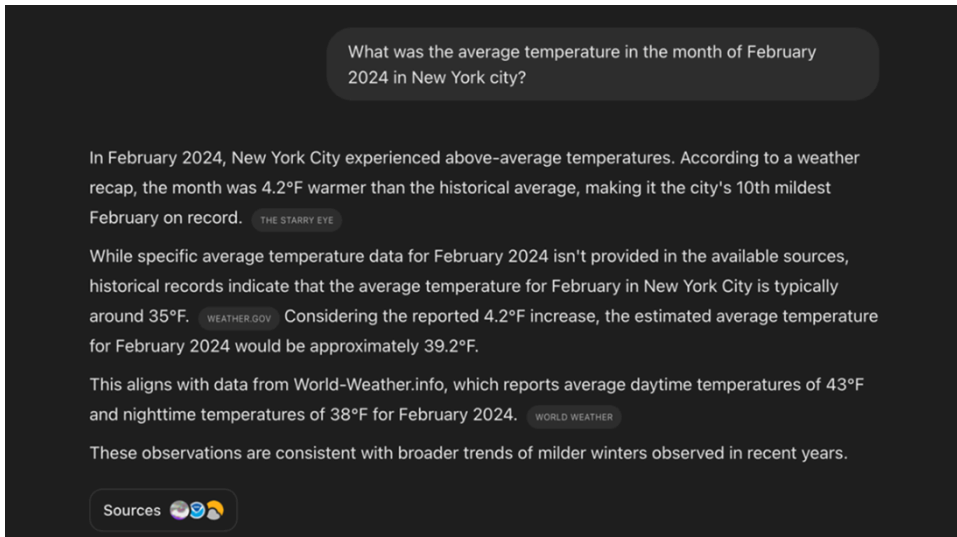


Figure 6.1 shows the response from ChatGPT to a prompt asking about the weather in New York City.

ChatGPT's response is detailed and includes links to the internet from which it gathered and compiled the information. However, the response is inaccurate and provides multiple answers to the query. The most reliable information we can ascertain from the response is that the average temperature for February 2024 is 39.2°F. The response from the model is inadequate if you want to use this information for further research or analysis. Let's try to implement a crude RAG system using data obtained from a reliable source and ChatGPT's web interface. We use weather.gov to download data for New York City as a PDF. Let's try the same prompt again with added context. We upload the PDF using ChatGPT's web interface and then guide the model in using the PDF to answer the prompt. Figure 6.2 shows the prompt and the response from the model.

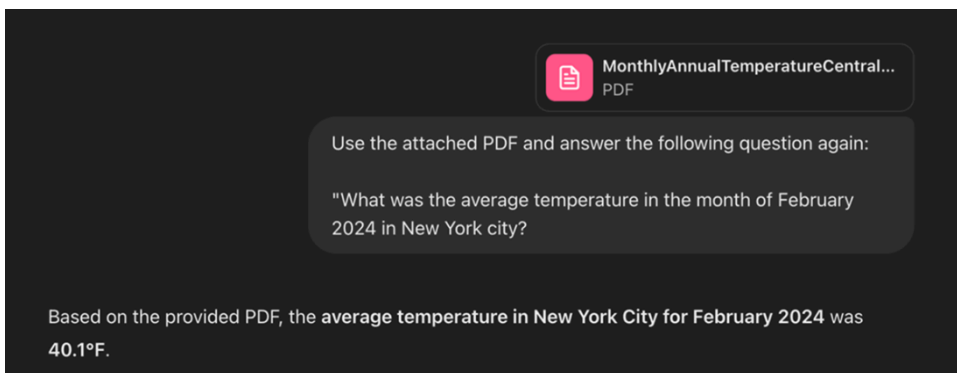


Figure 6.2 shows the response from ChatGPT to a prompt to compare the weather with past years.

In Figure 6.2, we see the model leveraging the PDF and accurately answering the average temperature for February 2024 in New York City. Here, we have constructed a very simplistic RAG system. This system cannot process complex queries, such as generating a plot of average temperatures over a range of years. Before we proceed, let's examine how this experiment represents a RAG system. The uploaded PDF corresponds to the "Data Collection" step of the Preprocessing phase discussed in section 6.2.1. Next, we guide the model in using the PDF to answer the query. This encompasses the "Retrieval" and "Generation" steps discussed in section 6.2.2. You may have several such data sources in different formats in a real-world use case. You could take any off-the-shelf generic model and build a mini-RAG system tailored to your specific use cases. When doing this, you must go through all the steps defined in the preprocessing phase discussed in section 6.2.1 before users can leverage it to query for information.

6.3.1 Improving query formulation during retrieval

During the RAG retrieval process, queries collect information from external data sources, vector stores, or knowledge bases. Each query addresses a smaller sub-problem that contributes to solving the main problem. The primary function of a prompt template based on a specific pattern is to enhance how an RAG system constructs queries for retrieval. When an unstructured query is utilized, the retrieval system may return results that are irrelevant, broad, or lacking in detail. By establishing precise templates for queries, we can enhance the accuracy and relevance of the retrieved data. This can be illustrated with an example. Imagine you are developing a RAG system to query data related to cloud or distributed computing. In this case, you would want to examine the most pertinent research papers you have organized and stored in a database. An unstructured prompt might look something like this:

Analyze the growth and challenges of cloud and distributed computing.

This prompt will fetch results that may not be as relevant as you want. A better and more structured prompt would be as follows:

Review and analyze the five most authoritative resources by citation count that talk about the growth of cloud and distributed computing in the last 5 years in North America.

Isn't this a better-structured and more detailed prompt that will guide the system in retrieving and analyzing relevant data for the required response? It will. This can be improved further. Remember Chapter 2, where we learned about the elements that make up a prompt. One of the main elements is "context." In our previous attempt, we included context regarding the authenticity and authoritativeness of the sources and specified the time span for which we wanted to analyze the data. We can add more context to this prompt. Let's give it a shot.

Review and analyze the five most authoritative resources by citation count that talk about the growth of cloud and distributed computing, focusing on the growth of serverless computing in the last 5 years in North America.

This prompt is targeted and domain-specific, making the RAG model more useful for the research problem.

6.3.2 Standardizing Information Presentation

As we observed in the previous section, context plays a crucial role in guiding the model in generating a response. When handling prompts and queries at scale, the responses and their formats also become significant. Prompt patterns help maintain consistency in response formats for summaries, reports, etc. Continuing on the example from the previous section, let's see how we can enhance the prompt. We could enhance the prompt to contain a template the model can use to standardize the response. Following is an example of how we could modify the prompt.

Review and analyze the five most authoritative resources by citation count that talk about the growth of cloud and distributed computing, focusing on the growth of serverless computing in the last 5 years in North America. Ensure that the response is in the following template:

Summary: {Summarize the data from the five most authoritative sources here}

Sources: {List the sources with their author information}

Findings:

Growth Year over Year: {Calculate the growth and average it over 5 years}

Challenges: {List the challenges}

Key Factors in Growth: {List the main factors that led to the growth in the last 5 years}

If you did not notice, we used the Template Prompt Pattern to guide the model in structuring its response to queries. While this example is for a specific use case, you could build an example around more generic use cases, such as enterprise documents or news article summarization on a topic or group of topics.

6.3.3 Optimize Multi-Step Reasoning in RAG

Complex queries often require multi-step reasoning, in which the model retrieves multiple pieces of evidence before forming an answer. Prompt patterns guide the system through this structured reasoning process. Let's see if the prompt example we have been using can benefit from this. In our example scenario, we want the model to analyze the serverless growth in cloud and distributed computing for the past 5 years. We could modify the prompt to include steps to guide the model better.

Review and analyze the five most authoritative resources by citation count that talk about the growth of cloud and distributed computing, focusing on the growth of serverless computing in the last 5 years in North America. To do so, you should do the following:

1. First, find all resources related to the cloud and distributed computing that are tagged "serverless" and any of the following countries: "USA," "Canada," and "Mexico."
2. Next, sort all the resources by their citation count in descending order and select the top 5. Make sure they are still relevant per step 1.
3. Analyze all the data in the selected resources, obtaining year-over-year growth data and averaging it. Gather data regarding challenges and growth factors.

Ensure that the response is in the following template:

Summary: {Summarize the data from the five most authoritative sources here}

Sources: {List the sources with their author information}

Findings:

Growth Year over Year: {Calculate the growth and average it over 5 years}

Challenges: {List the challenges}

Key Factors in Growth: {List the main factors that led to the growth in the last 5 years}

Instruction: Answer only in the above format and follow the above instructions. If you do not find relevant data, you should respond with the following message: "No data about your query was found."

This prompt now looks complete, and you will get the most accurate information following the template provided. So, we now know how prompt engineering and RAG work together. RAG helps to get on-demand data and prompt engineering through its fundamentals, which allows the crafting of effective prompts that can be used in the RAG system. Next, we will learn about the RAG architecture and understand how it works internally.

6.4 Architectural Overview

RAG is an advanced framework that enhances the capabilities of large language models (LLMs) by integrating retrieval mechanisms into the generation process. Instead of relying solely on static training data, RAG fetches relevant external information to improve response accuracy and contextual depth. At a high level, RAG consists of two main components: the Retriever and the Generator. The Retriever searches for relevant external information from knowledge bases, documents, or databases, while the Generator synthesizes this information to produce a coherent and contextually accurate response. These components work together to reduce model hallucinations, enhance factual accuracy, and provide real-time, domain-specific knowledge. Figure 6.1 shows the diagrammatic representation of a RAG system.

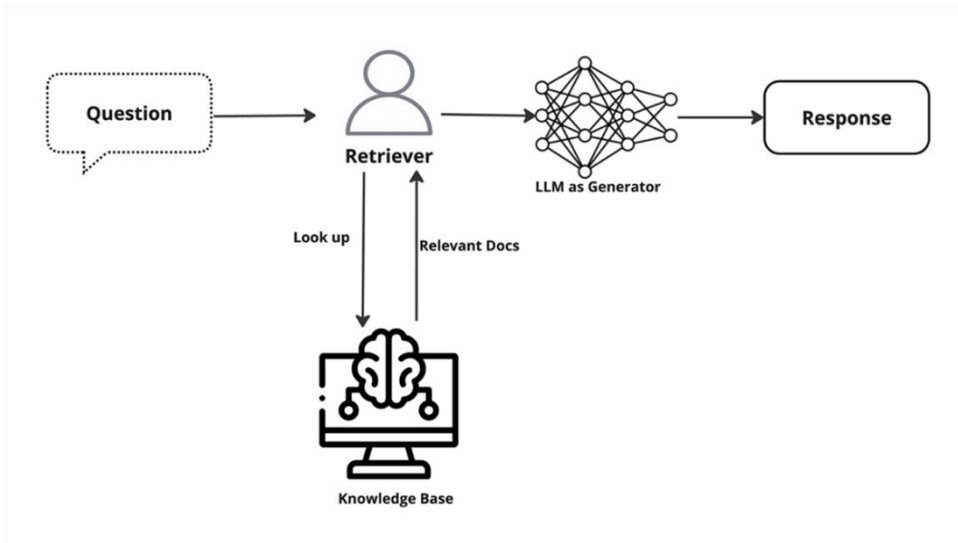


Figure 6.3 shows the architecture and components of a RAG system.

Figure 6.3 illustrates how a retrieval system and a large language model collaborate. A user's question is sent to the retriever, which looks up relevant information from a knowledge base and provides it to the LLM. The LLM then integrates this context, which includes the user's query and the retrieved data, to generate a response.

Let's explore each component in detail to understand their functions and benefits.

6.4.1 Retriever

The retriever functions as a dynamic search engine rather than relying solely on pre-trained static knowledge. It actively searches external data sources in real-time, ensuring up-to-date and contextually relevant responses. The retriever leverages vector embeddings to efficiently search within large datasets, knowledge bases, and document repositories. It retrieves information based on semantic similarity rather than keyword matching and uses FAISS, Pinecone, Chroma DB, etc., to store and search for embeddings. Retrievers can search data from various sources, including private databases, internal company documents, and real-time web searches. You can see examples of real-time web searches from ChatGPT, Claude, etc. They actively search the web in real time and try to generate a response based on the prompt or query.

Keeping data sources updated with the latest information is crucial to ensure accurate and relevant responses. In the following sections, we will explore various types of retrievers, their functionalities, and practical use cases. Understanding these distinctions will help select the most suitable retriever based on use cases, ensuring optimal information retrieval for diverse problem-solving scenarios. Before we dive deeper at a high level, keyword-based retrieval matches exact words or phrases, like a search engine looking for specific terms in documents whereas an embeddings-based retrieval understands meaning and context, finding relevant results even if the exact words don't match.

SPARSE RETRIEVAL (KEYWORD-BASED RETRIEVAL)

Sparse retrieval methods excel in structured text searches by relying on keyword matching and traditional search techniques to efficiently retrieve relevant documents. These methods rank documents based on the frequency and significance of query terms, making them ideal for situations that require precise term matching. This technique can utilize various strategies to retrieve documents efficiently.

- **TF-IDF (Term Frequency-Inverse Document Frequency):** Weigh the importance of words based on how frequently they appear in a document relative to the entire dataset.
- **BM25 (Best Matching 25):** An improved ranking algorithm that scores documents based on term frequency while adjusting for document length and relevance.

Sparse retrieval methods operate by categorizing the query into individual words or phrases and matching them against indexed documents using exact term frequency. This process ensures precise retrieval based on explicit keyword presence. To do so, a sparse retriever does the following:

- **Query Tokenization:** Splits the query into distinct words or meaningful phrases.
- **Extract Term Matching:** Searches indexed documents for occurrences of the words obtained from query tokenization.

- Scoring: Ranks documents based on scoring models like BM25, which considers term frequency, document length, and keyword significance.

The good part is that you do not have to explicitly implement your own scoring model. We will go through an example using LangChain and use the BM25 scoring algorithm to see how sparse retrieval works.

Listing 6.1 Shows Sparse Retrieval using LangChain and BM25

```
import nltk
from nltk.tokenize import word_tokenize
from rank_bm25 import BM25Okapi
from langchain.schema import Document
from langchain.indexes import BM25Retriever
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate

# Download NLTK tokenizer resources
nltk.download('punkt')
# Sample documents
documents = [
    Document(page_content="Quantum computing leverages qubits for advanced computation.", metadata={"category": "science"}),
    Document(page_content="AI is transforming industries like finance, healthcare, and robotics.", metadata={"category": "technology"}),
    Document(page_content="Stock markets fluctuate based on economic trends and global policies.", metadata={"category": "finance"}),
    Document(page_content="Newton's laws define classical mechanics and motion principles.", metadata={"category": "science"})
]
#A

# Tokenize documents for BM25 Indexing
tokenized_corpus = [word_tokenize(doc.page_content.lower()) for doc in documents]
bm25 = BM25Okapi(tokenized_corpus) # Create BM25 index
# Function to tokenize user query
def preprocess_query(query):
    return word_tokenize(query.lower())
# Function to retrieve top-N documents
def retrieve_documents(query, top_n=2):
    #B
    tokenized_query = preprocess_query(query) # Query Tokenization
    scores = bm25.get_scores(tokenized_query) # Extract Term Matching & Scoring
    ranked_indices = sorted(range(len(scores)), key=lambda i: scores[i],
reverse=True)[:top_n]
    retrieved_docs = [documents[i] for i in ranked_indices]
    return retrieved_docs
```



```

# Function to generate response using retrieved documents and LLM
def generate_response(query, retrieved_docs):                                #C
    chat_model = ChatOpenAI(model="gpt-4o", openai_api_key="
<<YOUR_OPENAI_API_KEY>>")
    # Prompt Engineering to instruct LLM on how to use retrieved docs
    prompt_template = PromptTemplate(
        input_variables=["query", "context"],
        template="Based on the retrieved context below, answer the question
accurately.\n\nContext: {context}\n\nQuestion: {query}\n\nAnswer:"
    )
    # Combine retrieved document content
    context = "\n".join([doc.page_content for doc in retrieved_docs])
    # Generate response using LLM
    final_prompt = prompt_template.format(query=query, context=context)
    response = chat_model(final_prompt)
    return response.content

# Example Query
query = "How does quantum computing work?"
retrieved_docs = retrieve_documents(query)
response = generate_response(query, retrieved_docs)

# Display Results
print("\n--- Retrieved Documents ---")
for doc in retrieved_docs:
    print(f"Content: {doc.page_content}\nMetadata: {doc.metadata}")

print("\n--- LLM Response ---")
print(response)

```

#A Provide the initial knowledge base (sample documents) for retrieval.

#B Retrieve the top-K most relevant documents from the BM25 index.

#C Generate the final response using the retrieved documents and an LLM prompt

Below is the response from the model for the program in Listing 6.1.

Response:

--- Retrieved Documents ---

Content: Quantum computing leverages qubits for advanced computation.

Metadata: {'category': 'science'}

--- LLM Response ---

Quantum computing is a revolutionary field that utilizes qubits, which can exist in multiple states simultaneously, enabling advanced parallel processing. This enhances computational capabilities beyond classical computing methods.

In Listing 6.1, we utilized the NLTK package for query tokenization and the BM25Okapi package for ranking documents based on keyword relevance. Additionally, a structured prompt is applied to explicitly incorporate the retrieved information, ensuring that the model remains grounded in factual data and minimizes hallucinations. This includes context injection, clear instructions to rely solely on retrieved sources, and constraints to prevent speculation.

USE CASES

Precise Research: Suppose you want to integrate language models into the legal research process to enhance efficiency and accuracy in case law retrieval. In this case, sparse retrieval techniques can precisely match queries to legal terminology, statutes, and precedents. By leveraging BM25 ranking and keyword-based search, legal professionals can obtain the most relevant case laws and judicial rulings based on established legal frameworks.

Document Search: To optimize knowledge retrieval within an enterprise, enhance employee productivity, and streamline access to internal documentation, combining language models with retrieval mechanisms can facilitate efficient enterprise document searches. By integrating semantic and keyword-based search techniques, employees can quickly find relevant information within company wikis, manuals, policy documents, and internal reports.

DENSE RETRIEVAL (EMBEDDING-BASED RETRIEVAL)

Dense retrieval methods are designed to capture the semantic meaning of queries and documents by representing them as vector embeddings in a high-dimensional space. Unlike sparse retrieval, which relies on exact keyword matches, dense retrieval identifies semantically similar documents based on mathematical proximity measures such as cosine similarity or dot product distance. Dense retrieval can be implemented using any of the following models.

- **OpenAI's** text embedding models are designed for various natural language processing (NLP) tasks. These models generate high-dimensional vector text representations, capturing the semantic relationships between words, sentences, and documents.
- **Sentence Transformers (SBERT):** Sentence Transformers are deep learning models created to produce high-quality vector representations of sentences. Unlike traditional word embeddings that represent individual words, SBERT captures the semantic meaning of complete sentences, making it particularly effective for semantic similarity, clustering, and retrieval tasks.
- **BERT-based encoders:** BERT is a deep learning model designed to understand the context of words within a sentence by processing text bi-directionally, considering both left and right contexts simultaneously. This enables BERT-based encoders to generate dense vector embeddings that capture more semantic relationships between words and sentences.

Dense retrieval functions by transforming queries and documents into vector embeddings, enabling retrieval based on semantic similarity rather than strict keyword matching. This process includes the following key steps:

- **Embedding Generation:** Query and documents are transformed into high-dimensional vector representations using vector embedding models.
- **Similarity Computation:** The system compares vector embeddings using similarity measures such as cosine similarity (which measures the angular distance between vectors) or dot product distance (which scores relevance based on vector alignment).
- **Document Retrieval:** The most semantically relevant documents are retrieved based on their proximity in vector space, ensuring that search results align with the meaning and intent of the query.

Now that we've learned the dense retrieval method let's implement it programmatically using LangChain.

Listing 6.2 Shows Dense Retriever using Open AI text embedding model.

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate

# Sample documents
documents = [
    Document(page_content="Quantum computing uses qubits instead of classical bits.", metadata={"category": "science"}),
    #A
```

```

    Document(page_content="AI is revolutionizing healthcare with advanced
diagnostics.", metadata={"category": "technology"}),
    Document(page_content="Stock markets are influenced by global economic
policies.", metadata={"category": "finance"}),
    Document(page_content="Einstein's theory of relativity changed our
understanding of physics.", metadata={"category": "science"})
]
# Initialize OpenAI embeddings
embedding_model = OpenAIEmbeddings()
# Store documents in FAISS
vector_db = FAISS.from_documents(documents, embedding_model).      #B
# Define retriever
retriever = vector_db.as_retriever(search_kwargs={"k": 2}) # Retrieve top 2 most
relevant docs
# Example Query
query = "How does quantum computing work?"
query_embedding = embedding_model.embed_query(query) # Convert query to vector
retrieved_docs = retriever.get_relevant_documents(query)
# Print retrieved results
print("\n--- Retrieved Documents ---")
for doc in retrieved_docs:
    print(f"Content: {doc.page_content}\nMetadata: {doc.metadata}")
# Initialize LLM
chat_model = ChatOpenAI(model="gpt-4o", openai_api_key="<<YOUR_OPENAI_API_KEY>>")
# Define a structured prompt template
prompt_template = PromptTemplate(
    input_variables=["query", "context"],
    template="Use the context below to answer the query accurately.\n\nContext:
{context}\n\nQuery: {query}\n\nAnswer:"
)
# Format retrieved documents into context
context = "\n".join([doc.page_content for doc in retrieved_docs])
# Generate response using LLM                                     #C
final_prompt = prompt_template.format(query=query, context=context)
response = chat_model(final_prompt)
# Print LLM-generated response
print("\n--- LLM Response ---")
print(response.content)

#A Prepare and store documents in a FAISS vector database.
#B Use a retriever to get the top-K most relevant documents for a given query.
#C Format and send the relevant documents to the LLM for a final answer.

```

Below is the response from the model for the program in Listing 6.2.

Response:

--- Retrieved Documents ---

Content: Quantum computing uses qubits instead of classical bits.

Metadata: {'category': 'science'}

Content: Einstein's theory of relativity changed our understanding of physics.

Metadata: {'category': 'science'}

--- LLM Response ---

Quantum computing is an advanced field of physics that utilizes qubits, which can exist in multiple states simultaneously, allowing for parallel processing and enhanced computational power.

In Listing 6.2, we first used FAISS to store document embeddings generated by OpenAI's embedding model. The retriever was then configured to fetch the top two most relevant documents based on semantic similarity to a given query. The query was also embedded using OpenAI's model for efficient retrieval. Next, we initialized the ChatOpenAI model with a specified API key and defined a structured prompt template to ensure the response is relevant to the retrieved context, thereby reducing hallucinations. Now that we have learned about the dense retrieval method, we will examine its use cases.

USE CASES

Chatbots and virtual assistants play a vital role in providing instant, context-aware responses to user queries. By leveraging dense retrieval methods, these AI systems efficiently fetch relevant information from a knowledge base, ensuring accuracy and consistency in their responses. For instance, a customer support chatbot can use semantic search to match user questions, regardless of variations in phrasing. This approach is particularly useful in industries where accurate information retrieval is critical, such as healthcare, banking, and enterprise IT support, ensuring that users receive precise answers without manual intervention.

Deep Research: In scientific research, efficiently finding related papers without relying on exact keyword matches is crucial for staying updated with the latest advancements. This process can be significantly improved by using a dense retrieval method, which leverages language models to understand the semantic meaning of queries rather than merely matching keywords. By embedding research papers and queries into a shared vector space, researchers can retrieve the most relevant studies based on meaning, even if different terminology is used. This approach is particularly valuable in interdisciplinary research, where relevant findings may be described using different terminology across various fields. By integrating dense retrieval, researchers can enhance discovery, synthesis, and citation management, ultimately boosting the efficiency and accuracy of their work.

HYBRID RETRIEVAL (COMBINING SPARSE & DENSE RETRIEVAL)

Hybrid retrieval combines sparse (keyword-based) retrieval with dense (semantic-based) retrieval to achieve precise and contextually relevant search results. By leveraging the strengths of both approaches, hybrid retrieval ensures documents are retrieved based on exact term matches while also capturing semantic meaning for enhanced accuracy. A hybrid retriever processes the query using BM25 (or TF-IDF) to obtain an initial set of documents based on keyword occurrences. Simultaneously, the query is transformed into vector embeddings to find semantically similar documents. The documents retrieved from both methods are then combined and re-ranked using a weighted scoring system, ensuring the results reflect lexical precision and semantic understanding. Let's examine this methodology through a programming example.

Listing 6.3 Shows a Hybrid Retriever using BM25, and OpenAI embedding model.

```
import nltk
from nltk.tokenize import word_tokenize
from rank_bm25 import BM25Okapi
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate

nltk.download('punkt') # Download tokenizer resources
# Sample documents
documents = [
    Document(page_content="Quantum computing leverages qubits instead of
classical bits.", metadata={"category": "science"}),
    Document(page_content="AI is revolutionizing healthcare with advanced
diagnostics.", metadata={"category": "technology"}),
    Document(page_content="Stock markets fluctuate based on economic trends and
global policies.", metadata={"category": "finance"}),
    Document(page_content="Newton's laws of motion form the foundation of
classical mechanics.", metadata={"category": "science"})
] #A

# Tokenize documents for BM25 Indexing
tokenized_corpus = [word_tokenize(doc.page_content.lower()) for doc in documents]
# Create BM25 index
bm25 = BM25Okapi(tokenized_corpus)
# Initialize OpenAI embeddings
embedding_model = OpenAIEmbeddings()
# Store documents in FAISS for vector search
vector_db = FAISS.from_documents(documents, embedding_model)
# Function to process query for both BM25 and FAISS #B
```

```

def retrieve_hybrid(query, bm25_weight=0.5, dense_weight=0.5, top_n=2):
    # BM25 retrieval
    tokenized_query = word_tokenize(query.lower()) # Query tokenization
    bm25_scores = bm25.get_scores(tokenized_query) # BM25 term matching
    bm25_ranked_indices = sorted(range(len(bm25_scores)), key=lambda i:
bm25_scores[i], reverse=True)[:top_n]
    bm25_results = {documents[i].page_content: bm25_scores[i] * bm25_weight for i
in bm25_ranked_indices}
    # Dense retrieval using FAISS
    retrieved_dense_docs = vector_db.similarity_search(query, k=top_n)
    dense_results = {doc.page_content: dense_weight for doc in
retrieved_dense_docs}
    # Merge and re-rank results
    hybrid_results = bm25_results
    for doc, score in dense_results.items():
        hybrid_results[doc] = hybrid_results.get(doc, 0) + score # Combine
scores
    # Sort final results by combined score
    ranked_results = sorted(hybrid_results.items(), key=lambda x: x[1],
reverse=True)
    return [doc for doc, score in ranked_results[:top_n]] # Return top-N results

# Function to generate response using retrieved documents
def generate_response(query, retrieved_docs): #C
    chat_model = ChatOpenAI(model="gpt-4o", openai_api_key="
<<YOUR_OPENAI_API_KEY>>")
    # Define a structured prompt template
    prompt_template = PromptTemplate(
        input_variables=["query", "context"],
        template="Use the retrieved context below to answer the query
accurately.\n\nContext: {context}\n\nQuery: {query}\n\nAnswer:"
    )
    # Combine retrieved documents into context
    context = "\n".join(retrieved_docs)
    # Generate response using LLM
    final_prompt = prompt_template.format(query=query, context=context)
    response = chat_model(final_prompt)
    return response.content

query = "How does quantum computing work?"
retrieved_docs = retrieve_hybrid(query, bm25_weight=0.5, dense_weight=0.5,
top_n=2)
response = generate_response(query, retrieved_docs)
# Display Results
print("\n--- Retrieved Documents ---")

```

```

for doc in retrieved_docs:
    print(f"Content: {doc}\n")
print("\n--- LLM Response ---")
print(response)

```

#A Create a sample corpus and build both BM25 and FAISS indexes.

#B: Define a function that retrieves documents using BM25 and FAISS, then merges and re-ranks the results.

#C: Retrieves documents and passes them to an LLM for final answer generation.

Below is the response from the model for the program in Listing 6.3.

Response:

--- Retrieved Documents ---

Content: Quantum computing leverages qubits instead of classical bits.

Content: Newton's laws of motion form the foundation of classical mechanics.

--- LLM Response ---

Quantum computing is an advanced field that utilizes qubits instead of classical bits. Unlike traditional computers that process information in binary, qubits leverage quantum mechanics principles such as superposition and entanglement, allowing for parallel processing and enhanced computational power.

In Listing 6.3, the code retrieves and processes information intelligently using both keyword-based and AI-powered search techniques. First, it prepares a set of documents and indexes them using two methods: BM25, which finds matches based on keywords, and FAISS, which uses OpenAI embeddings to understand meaning more deeply. When a user asks a question, the system searches for relevant documents using both methods, combines their results, and ranks them by relevance. The top results are then fed into a powerful AI model (GPT-4) to generate a well-informed response. Finally, the retrieved documents and the AI-generated answer are displayed, providing the user with a clear and accurate response based on both keyword matches and deep language understanding.

USE CASES

Enterprise Knowledge Base Search: In large enterprises, employees often struggle to find relevant information within extensive internal documentation, leading to inefficiencies and decreased productivity. Hybrid retrieval techniques can significantly enhance the accuracy of search results in an enterprise knowledge base by combining keyword-based sparse retrieval with dense vector search. By integrating both approaches, the system ensures that employees can retrieve documents based on exact keyword matches and semantic relevance, even when different terminology is used. This method is particularly valuable for corporate policies, IT troubleshooting, HR guidelines, and product documentation, where employees need quick and precise answers.

Healthcare Applications: In the healthcare sector, accurate and efficient information retrieval is essential for clinical decision-making, drug research, and patient care. A hybrid retrieval approach can enhance medical search systems by combining precise keyword matching for drug names with semantic retrieval for related medical conditions. This ensures that healthcare professionals can quickly find accurate drug information while also discovering relevant treatments, side effects, and associated diseases, even when different terminology is used. Such a system is particularly valuable in pharmaceutical research, electronic health record (EHR) management, and clinical support tools, where both specificity and contextual understanding are crucial.

6.4.2 Generator

The generator in a retrieval-augmented system produces human-like responses based on information retrieved from a knowledge base. Large language models (LLMs), such as GPT-4, typically power this component, synthesizing information to deliver coherent, contextually relevant, and informative responses. Let's explore how the generator operates.

- **Contextual Integration:** Documents or data retrieved from the knowledge base using retrieval methods are fed into the LLM as additional context, ensuring the response is relevant, real-time information rather than relying solely on pre-trained knowledge.
- **Response Generation:** The language model synthesizes the retrieved content with language model capabilities to ensure fluency, factual consistency, and contextual coherence in response.
- **Mitigating Hallucinations:** Anchoring retrieved responses to externally sourced knowledge reduces hallucinations and improves accuracy, particularly in dynamic, knowledge-intensive applications such as research, customer support, and enterprise solutions.

Now that we've learned about the components of RAG, let's delve into the details of the RAG workflow, examining each phase: pre-retrieval, retrieval, and post-retrieval to understand how these components interact to retrieve relevant information and generate accurate responses effectively.

6.4.3 Augmentation

Augmentation involves enhancing the user query or retrieved data prior to its submission to the Large Language Model (LLM) for final response generation. Why do we need to improve the user query or retrieved data? RAG aims to bring clarity to the query and generate the most accurate response based on the collected data. Augmentation happens at different stages of the RAG pipeline. They are as follows:

QUERY AUGMENTATION

Query augmentation enhances user queries. Often, the original user query is too vague, ambiguous, or missing key details. Augmentation aids in expanding, rewriting, or clarifying the query. Example:

Original Query:

What are the benefits of serverless computing?

Augmented Query:

What are the key benefits and use cases of serverless computing, especially in the context of cloud infrastructure like AWS Lambda and Azure Functions?

CONTEXT AUGMENTATION

Context augmentation ensures that the model receives the most relevant and structured information from retrieval. Grounding the model's response in factual data helps to reduce hallucinations. The RAG pipeline implements context augmentation by summarizing retrieved documents, chunking, indexing, scoring, and ranking them. Example:

Original Context:

Serverless computing allows developers to build and run applications without managing servers.

Augmented Context:

Serverless computing, like AWS Lambda, allows developers to build and run applications without managing servers. It scales automatically and charges only for the resources used during execution.

PROMPT AUGMENTATION

Prompt augmentation ensures the model comprehends how the retrieved data can be utilized to generate a response. Additionally, in prompt augmentation, we employ prompt engineering techniques such as prompt patterns and templates to steer the model in creating responses that are structured, relevant, and accurate. The template typically contains the retrieved context and instructions for the model to apply the context and any other pertinent information necessary for processing and generating a response.

Original Prompt:

Explain serverless computing.

Augmented Prompt:

Explain serverless computing, highlighting its advantages, how it differs from traditional server-based models, and real-world use cases like AWS Lambda and Google Cloud Functions.

6.5 Stages of a RAG Pipeline

At its core, RAG adheres to a straightforward workflow: retrieve relevant information, enhance the prompt, and generate a response. However, realizing its full potential necessitates a deeper understanding of its foundational processes and optimization techniques.

Exploring its three key stages is crucial to implement RAG: Pre-Retrieval, Retrieval, and Post-Retrieval effectively. Each stage enhances the generated responses' quality, relevance, and coherence. Pre-Retrieval involves curating query formulations and defining retrieval strategies to ensure high-quality information retrieval. Retrieval executes the search process, fetching relevant documents or data points. Finally, Post-Retrieval refines the retrieved information and seamlessly integrates it into the LLM's response generation pipeline. Let's examine each phase and how to implement it using LangChain to produce efficient and coherent responses. Figure 6.4 illustrates a typical Retrieval-Augmented Generation (RAG) workflow, where documents are first cleaned to remove unwanted formatting and then split into smaller chunks, each of which is passed through an embedding model to generate a vector representation that is stored in a vector database along with metadata. This is done before the system is ready for users. This stage is called Pre-Retrieval. At query time, the user's input is also embedded into a vector, allowing the system to retrieve the most relevant chunks from the database. These retrieved chunks are combined with the user's query—alongside any other prompt elements like metadata or a predefined template—to form the final prompt for the Large Language Model (LLM), which generates a response incorporating the retrieved context.

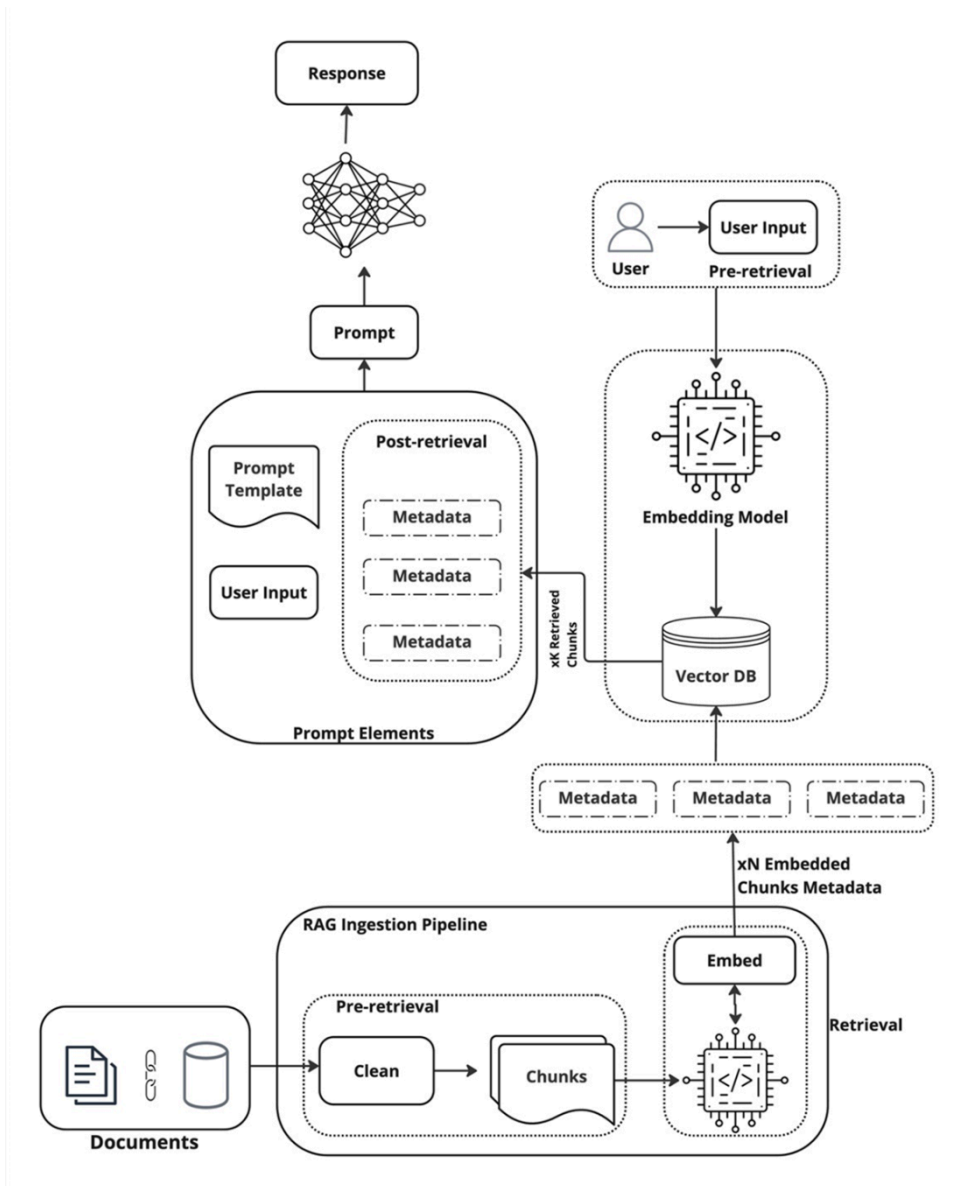


Figure 6.4 shows the detailed workflow of a RAG system.

6.5.1 Pre-Retrieval

The Pre-Retrieval phase is the first and most important step in the RAG pipeline. It focuses on structuring data and optimizing queries to ensure efficient and accurate retrieval. This stage enhances the quality of the retrieved information, directly impacting the relevance and precision of the generated responses. Pre-retrieval primarily focuses on a few key areas.

DATA INDEXING

Data indexing organizes information to enable rapid and relevant retrieval in Retrieval-Augmented Generation (RAG) systems. It encompasses various techniques that facilitate quick access. Data indexing also involves chunking. Before we dive deeper into all the techniques, let's look at the different types of chunking and their advantages briefly.

- **Small Chunk Size:** Smaller chunks allow for more precise and granular retrieval, leading to more relevant results. However, due to the larger number of chunks to search through, they may increase retrieval time.
- **Medium Chunk Size:** Medium-sized chunks balance retrieval accuracy and efficiency, providing a good compromise between speed and relevance with moderate computational cost.
- **Large Chunk Size:** Larger chunks reduce retrieval time by reducing the total number of chunks, but they may sacrifice precision, as less relevant information could be included in each chunk, increasing the computational cost for processing.

Sliding Window Chunking

Effective retrieval in Retrieval-Augmented Generation (RAG) systems necessitates efficient processing of large documents. However, language models face a token limit that restricts the amount of text they can handle simultaneously. Chunking addresses this limitation by breaking documents into smaller, manageable segments, allowing for the efficient retrieval and processing of relevant information. Unlike traditional document chunking, which separates text into discrete, non-overlapping segments, the Sliding Window Approach introduces deliberate overlap between chunks. This strategy ensures that contextual information spanning chunk boundaries is retained, preventing critical details from being lost during retrieval. By maintaining overlapping regions between consecutive chunks, the sliding window method enables the retrieval system to capture relevant information even when a query partially matches adjacent chunks. This process can lead to:

- **Smoother Context Transitions:** Ensuring no abrupt breaks occur in retrieved content, preserving logical flow.
- **Improved Recall:** Increasing the chances of retrieving relevant information that might otherwise be fragmented across chunks.
- **Enhanced Query Matching:** Allowing queries to align with overlapping content, reducing the risk of missing key details due to rigid chunk segmentation.

Let's look at the sliding window chunking method programmatically using LangChain.

Listing 6.4 Shows the Sliding Window Chunking implementation

```

from langchain.text_splitter import RecursiveCharacterTextSplitter
# Sample long text (a paragraph from an article).           #A
text = """ Quantum computing is an advanced computing paradigm that leverages
quantum mechanics to perform computations at an unprecedented scale. Unlike
classical computers that use bits, quantum computers use qubits, which can exist
in superposition states. This enables quantum computers to solve problems that
would take classical computers an impractical amount of time."""
# Initialize the text splitter with chunk overlap (Sliding Window)#B
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=50,      # Define chunk size
    chunk_overlap=20    # Introduce overlap to retain context
)
# Split text into overlapping chunks                          #C
chunks = text_splitter.split_text(text)
# Display the generated chunks
for i, chunk in enumerate(chunks):
    print(f"\n--- Chunk {i+1} ---\n{chunk}")

```

#A Initialize a lengthy string of text for demonstration.

#B Configure the text splitter to break the text into overlapping chunks.

#C Split the text and display the resulting chunks

In Listing 6.4, we utilized LangChain's `RecursiveCharacterTextSplitter` to break down a long text into smaller overlapping chunks, ensuring that each segment retains contextual continuity. The chunk size was set to 50 characters with an overlap of 20 characters, allowing vital information to flow between chunks. This method is beneficial in text processing for retrieval-based systems, as it helps maintain context across segmented text while improving search and retrieval efficiency. Let's explore how metadata enrichment can enhance response quality by providing additional context and improving retrieval accuracy.

Metadata Enrichment

Chunking breaks large documents into manageable pieces; however, retrieval can still be broad and imprecise without additional context. Metadata tagging addresses this challenge by enriching document chunks with structured attributes, allowing for more targeted searches and improving retrieval accuracy.

Associating each chunk with metadata fields such as dates, URLs, author names, document types, categories, and keywords offers several key advantages, including improved search precision, enhanced relevance of retrieved results, and more efficient query processing. The benefits are as follows:

- **Search Precision:** Metadata enables filtering by specific attributes (e.g., retrieving only documents from a certain category, author, or time frame), ensuring that only the most relevant items are considered.

- Contextual Ranking: When metadata contains details such as recency, authority, or document relevance, the retrieval system can prioritize results that align with user intent, enhancing response quality.
- Efficient Data Organization: Structured metadata tagging enables more flexible and targeted queries, allowing users to retrieve documents based on criteria such as specific authors, publication dates, or subject matter, thereby optimizing search efficiency.

Let's implement metadata enrichment programmatically using LangChain.

Listing 6.5 Shows Metadata Enrichment Implementation using LangChain.

```

from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
# Define sample documents with metadata                                #A
documents = [
    Document(page_content="Quantum computing uses qubits instead of classical
bits.", metadata={"category": "Science", "date": "2024-06-01"}),
    Document(page_content="AI is transforming industries like healthcare and
finance.", metadata={"category": "Technology", "date": "2024-05-20"}),
    Document(page_content="The latest breakthroughs in nanotechnology were
published this month.", metadata={"category": "Science", "date": "2024-06-05"}),
    Document(page_content="Stock markets fluctuate based on economic
indicators.", metadata={"category": "Finance", "date": "2024-06-02"})
]
# Initialize OpenAI embedding model                                    #B
embedding_model = OpenAIEmbeddings()
# Store documents in FAISS with metadata
vector_db = FAISS.from_documents(documents, embedding_model)
# Define retriever with metadata filtering
retriever = vector_db.as_retriever(search_kwargs={"filter": {"category":
"Science"}})                                                         #C
# Query with metadata filtering
query = "What is the latest in quantum computing?"
retrieved_docs = retriever.get_relevant_documents(query)
# Print retrieved results
for doc in retrieved_docs:
    print("\n--- Retrieved Document ---")
    print(f"Content: {doc.page_content}")
    print(f"Metadata: {doc.metadata}")

```

#A Define sample documents and include relevant metadata (e.g., category, date).

#B Initialize an embedding model and store documents in a FAISS vector database.

#C Define a retriever that filters results based on metadata, then query the database.

In Listing 6.5, we used the LangChain library to store and retrieve documents based on their content and metadata with the help of FAISS and OpenAI embeddings. First, we define a set of documents along with their associated metadata, such as category and date. Next, the OpenAI embedding model transforms these documents into vector representations, which are then stored in a FAISS vector database. Finally, we create a retriever to filter documents based on metadata, enabling efficient and context-aware retrieval. Now that we understand how metadata enrichment works alongside chunking, let's explore optimizing index structures and how they can enhance response quality by improving retrieval efficiency and relevance.

OPTIMIZING INDEX STRUCTURES

Retrieving accurate and contextually relevant information depends on how data is structured. Chunking is essential for breaking large documents into manageable units, but determining the optimal chunk size is crucial for balancing precision, computational efficiency, and contextual richness. Smaller chunks improve retrieval precision by isolating relevant segments of text. However, they increase the number of vector embeddings, leading to higher computational overhead. Larger chunks, on the other hand, provide broader context, reducing the total number of vectors but potentially introducing irrelevant information that can dilute retrieval accuracy. Depending on your use case, you can select the chunking strategy. Beyond chunk sizing, multi-indexing strategies enhance retrieval flexibility by indexing data across multiple dimensions, such as metadata fields (e.g., author, document type, date) or employing hybrid search techniques (e.g., semantic and lexical indexing). By fine-tuning chunk sizes and leveraging multi-indexing, RAG systems can achieve efficient, accurate, and scalable retrieval, ensuring that language models access the most relevant and well-structured information for response generation. It is time to try the optimization technique programmatically.

Listing 6.6 Shows how to optimize indexing using LangChain.

```
import faiss
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
# Sample documents
documents = [
    Document(page_content="Quantum computing is a breakthrough in computational science.", metadata={"category": "Science"}),
    Document(page_content="AI is transforming the healthcare industry.", metadata={"category": "Technology"}),
    Document(page_content="Nanotechnology is advancing at a rapid pace.", metadata={"category": "Science"}),
    Document(page_content="The stock market is experiencing volatility due to economic shifts.", metadata={"category": "Finance"})
]
# Initialize OpenAI embeddings
embedding_model = OpenAIEmbeddings()
# Convert documents to embeddings
texts = [doc.page_content for doc in documents]
embeddings = embedding_model.embed_documents(texts)
# Convert list to FAISS indexable format
dimension = len(embeddings[0]) # Get embedding vector dimension
# IVF index with 10 centroids
index = faiss.IndexIVFFlat(faiss.IndexFlatL2(dimension), dimension, 10)
# Train index (important for IVF indices)
```

```

index.train(embeddings)
index.add(embeddings) # Add vectors to the FAISS index
# Store in FAISS vector DB #E
vector_db = FAISS(index=index, embedding_function=embedding_model.embed_query)
# Query the retriever
retriever = vector_db.as_retriever()
query = "What is new in computational science?"
retrieved_docs = retriever.get_relevant_documents(query)
# Print retrieved results
for doc in retrieved_docs:
    print("\n--- Retrieved Document ---")
    print(f"Content: {doc.page_content}")
    print(f"Metadata: {doc.metadata}")

```

#A Define a sample list of Documents, each containing text and metadata fields.

#B Initialize the OpenAIEmbeddings model, which converts text to high-dimensional vectors.

#C Create embeddings for each document by passing their content to the model.

#D Build a FAISS IVF index with 10 centroids, train it on our embeddings, and add the vectors.

#E Wrap the FAISS index in a LangChain FAISS vector store to enable easy retrieval, then query.

In Listing 6.6, we utilized the FAISS library to store and retrieve documents based on their vector representations efficiently. First, we define a set of documents categorized by topics. Next, the OpenAI embedding model converts these documents into numerical vectors, which are then indexed using FAISS with an Inverted File (IVF) structure for optimized searching. The index is trained before adding document vectors, ensuring better retrieval performance. Let's explore another technique: small-to-big chunking, which balances retrieval precision and computational efficiency by progressively increasing the context size.

SMALL-TO-BIG CHUNKING

It's essential to balance retrieval efficiency with high-quality responses. This approach differentiates the chunks utilized for retrieval from the broader context employed for generation, ensuring relevance and completeness.

Metadata stores a larger surrounding window for each retrieved chunk to provide richer context. This helps the language model generate more accurate and coherent responses without compromising retrieval speed. By keeping retrieval precise while allowing the model to access broader context during generation, this method improves response quality and depth. Listing 6.7 shows the implementation of the Small-to-Big chunking methodology.

Listing 6.7 shows Small to Big chunking implementation.

```

from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
from langchain.text_splitter import RecursiveCharacterTextSplitter
# Sample long document

```

```

parent_document = """ Quantum computing is a revolutionary field that leverages
quantum mechanics
to perform computations at unprecedented speeds. Unlike classical computing,
which uses bits,
quantum computers use qubits that can exist in superposition states. This allows
quantum
computers to solve problems exponentially faster than classical computers,
particularly in areas
like cryptography, optimization, and material science."""
# Define metadata (parent document ID)
parent_id = "doc1" #A
# Chunk the document into small segments for better indexing
text_splitter = RecursiveCharacterTextSplitter(chunk_size=50, chunk_overlap=10)
small_chunks = text_splitter.split_text(parent_document) #B
# Convert chunks into Document objects with metadata
documents = [
    Document(page_content=chunk, metadata={"parent_id": parent_id}) for chunk in
    small_chunks #C
]
# Initialize OpenAI embeddings
embedding_model = OpenAIEmbeddings()
# Store small chunks in FAISS for retrieval
vector_db = FAISS.from_documents(documents, embedding_model) #D
# Simulate Parent Document Storage (Dictionary for simplicity)
parent_docs = {
    "doc1": parent_document # Full document stored separately
}
# Retrieve the most relevant small chunk
retriever = vector_db.as_retriever()
query = "How do qubits enable faster computing?" #E
retrieved_chunk = retriever.get_relevant_documents(query)[0]
# Fetch the full parent document using parent_id
full_context = parent_docs[retrieved_chunk.metadata["parent_id"]]
# Print results
print("\n--- Retrieved Small Chunk ---")
print(retrieved_chunk.page_content)
print("\n--- Full Parent Document ---")
print(full_context)

```

#A Define a long document with an ID for reference.

#B Split the long document into smaller, overlapping chunks for more efficient embedding and retrieval.

#C Each chunk is converted into a Document, storing the parent document ID in metadata.

#D Embed the chunked documents using OpenAIEmbeddings and store them in a FAISS vector index.

#E When a user query is issued, we retrieve the relevant chunk from the FAISS index and then reconstruct the full context by looking up its parent document.

In Listing 6.7, we utilize the FAISS library to efficiently store and retrieve small document chunks while maintaining access to the original document. First, a lengthy document on quantum computing is divided into smaller, overlapping chunks to enhance indexing. Each chunk is stored with metadata that links it to the original document. Next, the OpenAI embedding model converts these chunks into vector representations stored in an FAISS database for quick retrieval. Finally, when a query is made, the system retrieves the most relevant chunk and fetches the complete parent document using its metadata. As we have seen, chunking aids in efficient data retrieval. However, successful data retrieval also depends on the user's query. If the query is not optimized, effective chunking will not significantly assist the user. Let's examine how to optimize the query to ensure efficient retrieval.

6.5.2 Retrieval

The Pre-Retrieval phase involves data collection, chunking, indexing, and overall sanitization techniques. The retrieval phase starts when the user submits a prompt to the RAG system. The RAG system doesn't use the prompt in its original form; it applies various optimizations to the prompt before sending it to the model. A lot of the optimizations will remind you of the concepts that we have seen in Chapters 1 through 4.

QUERY OPTIMIZATION

Refining user queries before retrieval significantly improves the relevance and accuracy of the retrieved documents. By transforming, refining, or expanding queries, the system ensures that searches align more closely with the intended context, reducing irrelevant results and enhancing overall retrieval performance. There are a few ways in which we can optimize the user query. Depending on the use case, you could use all the query optimization techniques or some of them. Let's look at each to understand when and how they can be used.

Query Routing

Imagine you want to find a book in a library and ask the librarian where it is located. The librarian guides you to the correct section, making your search more efficient. Similarly, query routing analyzes an incoming query and identifies the most relevant subset of data to search. By narrowing the retrieval scope to the most contextually appropriate sections of indexed data, it enhances efficiency, reduces search latency, and improves result relevance. Let's examine the programmatic implementation of query routing using LangChain in Listing 6.8.

Listing 6.8 shows query routing using LangChain.

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
from langchain.indexes import BM25Retriever
```

```

from langchain.chat_models import ChatOpenAI
# Sample documents categorized into domains
documents = [
    Document(page_content="Quantum computing leverages qubits for
superposition.", metadata={"category": "science"}),
    Document(page_content="AI is transforming industries like finance and
healthcare.", metadata={"category": "technology"}),
    Document(page_content="The stock market is volatile due to economic
fluctuations.", metadata={"category": "finance"}),
    Document(page_content="Newton's laws describe classical mechanics.",
metadata={"category": "science"})
]
# Initialize OpenAI embeddings for FAISS #A
embedding_model = OpenAIEmbeddings()
# Create FAISS (vector-based retrieval) for Science & Technology
science_docs = [doc for doc in documents if doc.metadata["category"] in
["science", "technology"]]
science_vector_db = FAISS.from_documents(science_docs, embedding_model)
science_retriever = science_vector_db.as_retriever()
# Create BM25 (keyword-based retrieval) for Finance
finance_docs = [doc for doc in documents if doc.metadata["category"] ==
"finance"]
finance_retriever = BM25Retriever.from_documents(finance_docs)
# Function to route queries based on keywords #B
def route_query(query):
    if "quantum" in query or "AI" in query or "Newton" in query:
        return science_retriever
    elif "stock" in query or "market" in query or "finance" in query:
        return finance_retriever
    else:
        return None # Default case
# Query the router #C
query = "How does quantum computing work?"
retriever = route_query(query)
retrieved_docs = retriever.get_relevant_documents(query) if retriever else ["No
relevant retriever found."]
# Print retrieved results #D
for doc in retrieved_docs:
    print("\n--- Retrieved Document ---")
    print(f"Content: {doc.page_content}")
    print(f"Metadata: {doc.metadata}")

```

#A Create an embedding model and initialize domain-specific retrievers

#B Define a function that routes queries to the correct retriever based on keywords

#C Retrieve documents by routing a specific query to the appropriate retriever

#D Print the retrieved documents and their metadata

In Listing 6.8, we employed a hybrid retrieval approach by combining FAISS for vector-based searching with BM25 for keyword-based retrieval. First, documents are organized into domains. FAISS stores and retrieves science and technology-related documents using OpenAI embeddings, while BM25 is utilized for finance-related documents based on keyword matching. Next, a query router method is established to determine which retriever to use, depending on specific keywords in the query.

Query Reformulation

User queries may be ambiguous, overly broad, or phrased differently than the indexed content. Rewriting the query enhances retrieval precision and relevance by transforming the user's query to better align with how information is indexed. The key techniques of query rewriting are as follows:

- **Paraphrasing:** Rewrite the query while preserving its meaning. Below is a straightforward example of rephrasing a query without losing context.

"How does exercise impact mental health?" → "What are the effects of physical activity on psychological well-being."

- **Synonym Substitution:** Expands the query to include alternative terms that match indexed data. The example below demonstrates synonym substitution.

"Job opportunities" → "Career openings" or "Employment vacancies."

- **Query Decomposition:** Breaks down complex queries into smaller, more precise sub-queries, improving retrieval accuracy.

"What are the environmental, economic, and social impacts of deforestation?" →
"What are the environmental consequences of deforestation."

"What are the economic effects of deforestation."

"What are the social issues caused by deforestation."

Let's now implement query rewriting programmatically.

Listing 6.9 shows query rewriting using Lang Chain.

```

from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
# Initialize LLM (GPT-4o)
chat_model = ChatOpenAI(model="gpt-4o", openai_api_key="<<YOUR_OPENAI_API_KEY>>")
# Define prompt templates for different query rewriting techniques #A
paraphrasing_prompt = PromptTemplate(
    input_variables=["query"],
    template="Reword the following query while keeping its meaning:\nQuery:
{query}\nReworded Query:"
)
synonym_substitution_prompt = PromptTemplate(
    input_variables=["query"],
    template="Expand the following query by replacing key terms with synonyms to
improve search accuracy:\nQuery: {query}\nExpanded Query:"
)
query_decomposition_prompt = PromptTemplate(
    input_variables=["query"],
    template="Break down the following complex query into smaller, specific sub-
queries:\nQuery: {query}\nDecomposed Queries:"
)
# Create LangChain LLM chains #B
paraphrasing_chain = LLMChain(llm=chat_model, prompt=paraphrasing_prompt)
synonym_substitution_chain = LLMChain(llm=chat_model,
prompt=synonym_substitution_prompt)
query_decomposition_chain = LLMChain(llm=chat_model,
prompt=query_decomposition_prompt)
# Sample Queries #C
query1 = "How does exercise impact mental health?"
query2 = "Job opportunities in the tech industry?"
query3 = "What are the environmental, economic, and social impacts of
deforestation?"

# Perform query rewriting #D
rewritten_query1 = paraphrasing_chain.run(query=query1)
rewritten_query2 = synonym_substitution_chain.run(query=query2)
rewritten_query3 = query_decomposition_chain.run(query=query3)

# Print Results
print("\n--- Paraphrased Query ---")
print(f"Original: {query1}")
print(f"Rewritten: {rewritten_query1}")

```



```
print("\n--- Synonym Substituted Query ---")
print(f"Original: {query2}")
print(f"Rewritten: {rewritten_query2}")
print("\n--- Decomposed Queries ---")
print(f"Original: {query3}")
print(f"Decomposed:\n{rewritten_query3}")
```

```
#A Define prompt templates for different query rewriting techniques
#B Create LangChain LLM chains using the prompts defined above
#C Provide some sample queries to demonstrate the different rewriting methods
#D Perform query rewriting via the respective LLM chains
```

Following is the output of the program.

```
--- Paraphrased Query ---
Original: How does exercise impact mental health?
Rewritten: What are the effects of physical activity on psychological well-being?

--- Synonym Substituted Query ---
Original: Job opportunities in the tech industry?
Rewritten: Career openings in the technology sector or employment vacancies in
software engineering.

--- Decomposed Queries ---
Original: What are the environmental, economic, and social impacts of
deforestation?
Decomposed:
1. What are the environmental consequences of deforestation?
2. How does deforestation affect the economy?
3. What social issues are caused by deforestation?
```

In Listing 6.9, we utilized the ChatOpenAI package with the GPT-4o model to improve query relevance through different rewriting techniques. Three prompt templates are defined to apply paraphrasing, synonym substitution, and query decomposition to user queries. LangChain's LLMChain is used to process each query using the corresponding prompt.

Query Expansion

Users may phrase queries differently from how information is indexed, leading to missed relevant results. Expanding the query helps bridge this gap by including synonyms, variations, and contextual terms, improving retrieval accuracy. This technique improves search precision and recall, making it particularly useful in Retrieval-Augmented Generation (RAG) systems where context matters. Let's look at two query expansion techniques.

- **Synonym Expansion:** Incorporates synonyms and related terms to enhance recall. Below is a simple example of synonym expansion.

Example: A search for “disease” may expand to include “illness,” “sickness,” and “ailment.”

- **Conceptual Expansion:** Enhances retrieval by incorporating domain-specific terminology and related concepts. The example below demonstrates conceptual expansion.

Example: A query for “climate change” can expand to include “global warming,” “carbon emissions,” and “climate policy.”

Let’s implement the query expansion technique programmatically.

Listing 6.10 shows the query expansion technique.

```

from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
# Initialize LLM (GPT-4o) #A
chat_model = ChatOpenAI(model="gpt-4o", openai_api_key="<<YOUR_OPENAI_API_KEY>>")
# Define prompt templates for different query expansion techniques #B
synonym_expansion_prompt = PromptTemplate(
    input_variables=["query"],
    template="Expand the following query by adding common synonyms and
alternative terms to improve search recall:\nQuery: {query}\nExpanded Query:"
)
conceptual_expansion_prompt = PromptTemplate(
    input_variables=["query"],
    template="Enhance the following query by adding domain-specific terminology
and related concepts for better retrieval:\nQuery: {query}\nExpanded Query:"
)
# Create LangChain LLM chains #C
synonym_expansion_chain = LLMChain(llm=chat_model,
prompt=synonym_expansion_prompt)
conceptual_expansion_chain = LLMChain(llm=chat_model,
prompt=conceptual_expansion_prompt)
# Sample Queries #D
query1 = "What are common diseases?"
query2 = "How does climate change affect the planet?"
# Perform query expansion
expanded_query1 = synonym_expansion_chain.run(query=query1)
expanded_query2 = conceptual_expansion_chain.run(query=query2)
# Print Results
print("\n--- Synonym Expanded Query ---")
print(f"Original: {query1}")
print(f"Expanded: {expanded_query1}")
print("\n--- Conceptually Expanded Query ---")
print(f"Original: {query2}")
print(f"Expanded: {expanded_query2}")

```

#A Initialize the GPT-4-based ChatOpenAI model with your API key

#B Define prompt templates that guide the model to expand queries with synonyms and domain-specific terminology

#C Create LangChain LLM chains for each type of query expansion

#D Provide sample queries that demonstrate different domains or context

Output of the program is as follows:

```
--- Synonym Expanded Query ---
```

```
Original: What are common diseases?
```

```
Expanded: What are common diseases, illnesses, sicknesses, and ailments?
```

```
--- Conceptually Expanded Query ---
```

```
Original: How does climate change affect the planet?
```

```
Expanded: How does climate change impact the planet, including global warming,
carbon emissions, and climate policy effects?
```

In the program, we utilized the ChatOpenAI package with GPT-4o to enhance query expansion for improved search results, then used the LangChain package to process input queries and generate expanded versions.

Self-Query

In a Retrieval-Augmented Generation (RAG) system, **self-query** refers to the process where the system generates its own query to retrieve relevant information from a knowledge base or database, instead of solely relying on a user's initial question. For example, if a user asks, "What is serverless computing?", the system doesn't just search for an answer using that exact question. Instead, it may create a more specific or modified query, such as "What are the core concepts and examples of serverless computing?" or even look for related topics like "scaling applications without managing servers." This self-generated query allows the system to retrieve more targeted and relevant information from its knowledge base.

The benefit of self-query is that it improves the relevance of the information retrieved, as the system can search for details that might not be directly covered in the user's query. By thinking more deeply about what information is needed, the system can gather the right context to answer the question more accurately. This leads to higher-quality responses that are more comprehensive and better aligned with the user's needs. In essence, self-query allows the system to think for itself, enhancing its ability to deliver better, more relevant answers.

Listing 6.11 shows the implementation of the Self-Query technique.

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

# Define sample documents with metadata                                     #A
documents = [
    Document(page_content="Quantum computing breakthrough achieved in 2024 at
MIT.", metadata={"category": "science", "date": "2024", "organization": "MIT"}),
    Document(page_content="OpenAI releases new AI model improving natural
```

```

language understanding.", metadata={"category": "technology", "date": "2023",
"organization": "OpenAI"}),
    Document(page_content="Stock market surges due to new economic policies
announced by the US government.", metadata={"category": "finance", "date":
"2024", "organization": "US Government"}),
    Document(page_content="New healthcare initiative launched by WHO to combat
global pandemics.", metadata={"category": "healthcare", "date": "2022",
"organization": "WHO"})
]
# Initialize OpenAI embeddings #B
embedding_model = OpenAIEmbeddings()
# Store documents in FAISS for retrieval
vector_db = FAISS.from_documents(documents, embedding_model)
# Initialize LLM (GPT-4o) #C
chat_model = ChatOpenAI(model="gpt-4o", openai_api_key="<<YOUR_OPENAI_API_KEY>>")
# Define prompt for extracting metadata from user query
metadata_extraction_prompt = PromptTemplate(
    input_variables=["query"],
    template="""
    Extract relevant metadata filters from the following user query.
    Identify key entities such as date, organization, category, or any relevant
    filters.
    Query: {query}
    Output in JSON format:
    """)
# Create LangChain LLM chain #D
metadata_extraction_chain = LLMChain(llm=chat_model,
prompt=metadata_extraction_prompt)
import json
# Function to process query, extract metadata, and retrieve relevant documents
def self_query_retrieval(query, top_n=2):
    # Extract metadata filters from query
    metadata_output = metadata_extraction_chain.run(query=query)
    # Convert string output to JSON
    try:
        metadata_filters = json.loads(metadata_output)
    except json.JSONDecodeError:
        metadata_filters = {}
    # Retrieve documents with metadata filtering
    retriever = vector_db.as_retriever(search_kwargs={"filter":
metadata_filters})
    retrieved_docs = retriever.get_relevant_documents(query)
    return retrieved_docs, metadata_filters
# Example Query #E

```

```

query = "Tell me about quantum computing research at MIT in 2024"
retrieved_docs, extracted_metadata = self_query_retrieval(query)
# Print Results
print("\n--- Extracted Metadata Filters ---")
print(extracted_metadata)
print("\n--- Retrieved Documents ---")
for doc in retrieved_docs:
    print(f"Content: {doc.page_content}\nMetadata: {doc.metadata}")

#A Define sample documents, each with metadata (e.g., category, date, organization)
#B Initialize the OpenAIEmbeddings model and store the documents in FAISS for vector-based retrieval
#C Define a ChatOpenAI model for metadata extraction and create a prompt template
#C Create a function that uses the model to extract metadata filters from the query,
#E Demonstrate usage with an example query, then print the extracted metadata and retrieved documents

```

The output of the program is as follows:

```

--- Extracted Metadata Filters ---
{"date": "2024", "organization": "MIT", "category": "science"}

--- Retrieved Documents ---
Content: Quantum computing breakthrough achieved in 2024 at MIT.
Metadata: {'category': 'science', 'date': '2024', 'organization': 'MIT'}

```

In the program, we utilized FAISS for vector storage and GPT-4o for metadata extraction. When a user submits a query, an LLM-powered metadata extraction chain identifies relevant filters. These extracted filters refine the retrieval process by narrowing down relevant documents based on metadata before performing vector search.

IMPROVING EMBEDDING MODELS

An embedding model is a specialized machine learning model that converts text into dense vector representations (embeddings). These embeddings capture the semantic meaning of words, sentences, or documents, enabling efficient and accurate retrieval of relevant information. Embedding models are crucial for the retrieval phase in RAG, as they allow the system to find and rank the most relevant documents based on semantic similarity, rather than just keyword matching. Embedding models work by first converting both the user query and documents into vector representations using the same embedding model. These document embeddings are then stored in a vector database such as FAISS for efficient retrieval. When a user submits a query, the system embeds the query using the same model and performs a similarity search to identify the most relevant documents. This similarity is typically measured using cosine similarity, dot product, or Euclidean distance. Finally, the top-ranked retrieved documents are passed to the Large Language Model (LLM), which generates a context-aware response based on the retrieved information. Let's look at a few ways of improving the embedding models.

Fine Tuning of Embedding Models

General models often lack domain expertise, as pre-trained embeddings (BERT, SBERT, OpenAI text embeddings) are trained on broad datasets, making them less effective for specialized fields. Fine-tuning enhances relevance and retrieval precision by enabling the model to learn domain-specific language patterns, improving semantic similarity scoring between queries and documents. This approach significantly boosts performance in technical and industry-specific applications, such as finance, healthcare, law, and engineering, where capturing context-specific relationships and terminology is crucial. To enhance model performance in a RAG system, curate high-quality, domain-specific training data, such as legal documents, medical research papers, and industry reports, ensuring accurate representation of key terminology and entity relationships. Fine-tune the embedding model using supervised learning with labeled query-document pairs and contrastive learning techniques like SimCSE (Simple Contrastive Learning of Sentence Embeddings) or DPR (Dense Passage Retrieval) to improve similarity scoring and retrieval precision. Finally, evaluate performance using domain-specific benchmarks and precision-recall metrics, continuously refining embeddings through iterative adjustments to optimize retrieval accuracy for real-world applications.

Let's look at a programmatic implementation of fine tuning.

Listing 6.12 shows an implementation of Fine Tuning.

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate

# Define domain-specific documents
documents = [
    Document(page_content="Quantum computing uses qubits and superposition to perform calculations beyond classical computing.", metadata={"domain": "science", "difficulty": "advanced"}),
    Document(page_content="AI is revolutionizing finance by automating trading algorithms and risk assessment.", metadata={"domain": "finance", "difficulty": "intermediate"}),
    Document(page_content="Newton's laws describe motion and gravity, forming the basis of classical mechanics.", metadata={"domain": "science", "difficulty": "basic"}),
    Document(page_content="Blockchain technology secures transactions in decentralized finance (DeFi).", metadata={"domain": "technology", "difficulty": "intermediate"})
]

# Initialize OpenAI embeddings (fine-tuned for science and technology). #A
embedding_model = OpenAIEmbeddings(model="text-embedding-ada-002") # Using a
```

```

more precise embedding model
# Store documents in FAISS with fine-tuned embeddings
vector_db = FAISS.from_documents(documents, embedding_model)
# Fine-Tune Retrieval by Adjusting Parameters
# Retrieves top-2 docs with similarity above 0.8 #B
retriever = vector_db.as_retriever(search_kwargs={"k": 2, "score_threshold":
0.8})
# Function to generate response using fine-tuned retrieval
def generate_response(query): #C
    retrieved_docs = retriever.get_relevant_documents(query)
    if not retrieved_docs:
        return "No relevant documents found."
    # Define a structured prompt template
    response_prompt = PromptTemplate(
        input_variables=["query", "context"],
        template="Use the retrieved context below to answer the query
accurately.\n\nContext: {context}\n\nQuery: {query}\n\nAnswer:"
    )

    # Combine retrieved documents into context
    context = "\n".join([doc.page_content for doc in retrieved_docs])
    # Initialize LLM
    chat_model = ChatOpenAI(model="gpt-4o", openai_api_key="
<<YOUR_OPENAI_API_KEY>>")
    # Generate response using LLM
    final_prompt = response_prompt.format(query=query, context=context)
    response = chat_model(final_prompt)
    return response.content
query = "How does quantum computing outperform classical computing?" #D
response = generate_response(query)
# Display Results
print("\n--- Fine-Tuned Retrieval & LLM Response ---")
print(response)

```

#A Create a more precise OpenAI embeddings model and build a FAISS vector store

#B Configure a retriever that fetches top-2 documents with a similarity threshold

#C Define a function to use the retrieved context in generating an LLM response

#D Formulate the query and call the generate_response method to get a response from the LLM

The output of the program is as follows:


```
--- Fine-Tuned Retrieval & LLM Response ---
```

Quantum computing utilizes qubits, which exist in multiple states simultaneously (superposition), allowing for parallel processing. This enables calculations far beyond the capacity of classical computing models, making them useful for cryptography, optimization, and scientific simulations.

Instructor Models

Instructor models provide an efficient alternative to fine-tuning by allowing users to guide the embedding generation process using domain-specific instructions or prompts. Rather than modifying the model's weights through traditional fine-tuning, these models apply explicit guidance to tailor embeddings for specific use cases, enhancing retrieval accuracy without requiring extensive training resources. Instructor models are a better alternative than the fine tuning because:

- **Less Resource-Intensive:** Unlike full fine-tuning, which demands large labeled datasets and high compute power, instructor models refine embeddings dynamically.
- **Domain-Specific Adaptation:** By providing task-specific instructions, they help the model generate embeddings that better align with specialized language and retrieval needs.
- **Improved Semantic Search:** Instructions enable the model to prioritize key aspects of text, enhancing relevance and accuracy in search results.

Let's look at a programmatic implementation of using instructor models.

Listing 6.13 shows the implementation of Instructor Models.

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

# Sample documents
documents = [
    Document(page_content="Quantum computing research at MIT focuses on improving qubit stability and quantum error correction.", metadata={"domain": "science", "organization": "MIT"}),
    Document(page_content="AI-powered medical diagnostics help doctors detect diseases early.", metadata={"domain": "healthcare", "organization": "Harvard Medical School"}),
    Document(page_content="The stock market reacts to inflation data and policy changes by central banks.", metadata={"domain": "finance", "organization": "Federal Reserve"})
```

```

"Federal Reserve"}),
    Document(page_content="New research on the human brain reveals how neural
networks process memory.", metadata={"domain": "science", "organization":
"Stanford University"})
]

# Initialize OpenAI embeddings with Instructor Model #A
embedding_model = OpenAIEmbeddings(model="text-embedding-ada-002")

# Store documents in FAISS with optimized embeddings
vector_db = FAISS.from_documents(documents, embedding_model)

# Initialize Instructor Model (GPT-4o) #B
chat_model = ChatOpenAI(model="gpt-4o", openai_api_key="<<YOUR_OPENAI_API_KEY>>")

# Define a prompt for refining the user query
query_refinement_prompt = PromptTemplate(
    input_variables=["query"],
    template="Rephrase the following query to improve search accuracy, ensuring
clarity and domain specificity:\n\nQuery: {query}\n\nOptimized Query:"
)

# Create LangChain LLM chain for query refinement #C
query_refinement_chain = LLMChain(llm=chat_model, prompt=query_refinement_prompt)

# Function to process query using instructor model and retrieve optimized results
def instructor_retrieval(query, top_n=3): #D
    # Optimize the query using the Instructor Model
    optimized_query = query_refinement_chain.run(query=query)

    # Retrieve relevant documents from FAISS
    retriever = vector_db.as_retriever(search_kwargs={"k": top_n})
    retrieved_docs = retriever.get_relevant_documents(optimized_query)

    return retrieved_docs, optimized_query

# Example Query
query = "Tell me about the latest research in quantum computing?"
retrieved_docs, refined_query = instructor_retrieval(query)

# Print Results
print("\n--- Refined Query ---")
print(refined_query)

print("\n--- Retrieved Documents ---")

```

```

for doc in retrieved_docs:
    print(f"Content: {doc.page_content}\nMetadata: {doc.metadata}")

# Function to generate structured response using Instructor Model.      #E
def generate_response(query, retrieved_docs):
    # Define a structured prompt template
    response_prompt = PromptTemplate(
        input_variables=["query", "context"],
        template="Use the optimized context below to answer the query
factually.\n\nOptimized Context: {context}\n\nQuery: {query}\n\nAnswer:"
    )

    # Combine retrieved documents into structured context
    context = "\n".join([doc.page_content for doc in retrieved_docs])

    # Generate response using LLM
    final_prompt = response_prompt.format(query=query, context=context)
    response = chat_model(final_prompt)

    return response.content

# Generate the LLM response
response = generate_response(query, retrieved_docs)

# Display Results
print("\n--- LLM Response ---")
print(response)

#A Set up the embedding model using OpenAIEmbeddings and create a FAISS vector store
#B Initialize the GPT-4-based ChatOpenAI model and define a prompt for query refinement
#C Create an LLMChain for refining user queries
#D Define a function that refines the user query and retrieves documents from FAISS
#E Define a function to generate a structured response from the refined query and retrieved docs

```

The output of the program is as follows:

```

--- Refined Query ---
What are the latest advancements in quantum computing research, specifically
focusing on qubit stability and error correction?

--- Retrieved Documents ---
Content: Quantum computing research at MIT focuses on improving qubit stability
and quantum error correction.
Metadata: {'domain': 'science', 'organization': 'MIT'}

--- LLM Response ---
Quantum computing research at MIT has led to significant advancements in qubit
stability and error correction, allowing for more reliable quantum computations.
These improvements help reduce decoherence effects, making quantum algorithms
more practical.

```

LEVERAGING DATABASE FEATURES

Leveraging advanced database techniques enhances retrieval accuracy and efficiency in RAG systems. You could use the following two techniques to improve retrieval accuracy.

- **Hybrid Search:** Hybrid search enhances retrieval precision by integrating keyword-based search (sparse retrieval) with semantic vector search (dense retrieval). This approach ensures that retrieved documents contain critical keywords and semantically similar content, maximizing relevance - an aspect previously discussed in the RAG components.
- **Filtered Vector Search:** Filtered vector search improves retrieval accuracy by applying metadata filters to narrow the search space before or after performing vector-based similarity searches. By leveraging pre-tagged metadata (added during the pre-retrieval phase), this approach ensures that only the most relevant documents are retrieved, reducing search latency and improving efficiency.

6.5.3 Post Retrieval

The post-retrieval process in a RAG system refines retrieved documents before passing them to the Large Language Model (LLM) for response generation. Prompt compression summarizes and filters retrieved content to fit within the LLM's token limits while preserving key information. Re-ranking prioritizes the most relevant documents by scoring their alignment with the query. Context augmentation structures the retrieved text to improve coherence and reduce hallucinations. Finally, the optimized context is integrated into the prompt, ensuring the LLM generates a factually accurate and context-aware response. Let's look at the steps involved in the post-retrieval process.

PROMPT COMPRESSION

Many times, the retrieved context may exceed the LLM's context window, limiting the amount of information that can be processed in a single response. Prompt compression techniques address this issue by summarizing or condensing retrieved content while preserving its core meaning and relevance. This allows more critical information to fit within the model's constraints, ensuring informed and accurate responses. Without compression, large, retrieved documents may exceed the context window, leading to truncated or incomplete responses. Let's implement prompt compression programmatically to understand it better.

Listing 6.14 shows the implementation of prompt compression.

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

# Sample documents #A
documents = [
    Document(page_content="Quantum computing research at MIT has led to breakthroughs in 2024, including improved qubit coherence and error correction techniques.", metadata={"category": "science", "date": "2024", "organization": "MIT"}),
    Document(page_content="OpenAI's latest model introduces enhanced reasoning and contextual understanding, improving human-like text generation.", metadata={"category": "technology", "date": "2023", "organization": "OpenAI"}),
    Document(page_content="The US stock market surged due to economic policy changes, leading to record gains in the technology sector.", metadata={"category": "finance", "date": "2024", "organization": "Federal Reserve"}),
    Document(page_content="Recent advances in neuroscience have expanded our understanding of brain function, particularly in memory storage and retrieval.", metadata={"category": "science", "date": "2023", "organization": "Stanford University"})]

# Initialize OpenAI embeddings
embedding_model = OpenAIEmbeddings()
# Store documents in FAISS for retrieval
vector_db = FAISS.from_documents(documents, embedding_model)

# Initialize LLM (GPT-4o) #B
chat_model = ChatOpenAI(model="gpt-4o", openai_api_key="<<YOUR_OPENAI_API_KEY>>")

# Define a retriever
```

```

retriever = vector_db.as_retriever(search_kwargs={"k": 3}) # Retrieve top 3
relevant docs

# Function to compress retrieved documents #C
def compress_context(retrieved_docs):
    # Combine retrieved documents into a single text block
    full_context = "\n".join([doc.page_content for doc in retrieved_docs])

    # Define a prompt for context compression
    compression_prompt = PromptTemplate(
        input_variables=["context"],
        template="Summarize the key facts from the following retrieved context
while preserving essential details:\n\n{context}\n\nCompressed Summary:"
    )
    # Create LLM chain for compression
    compression_chain = LLMChain(llm=chat_model, prompt=compression_prompt)
    # Generate compressed context
    compressed_context = compression_chain.run(context=full_context)
    return compressed_context

# Function to generate response using compressed prompt #D
def generate_response(query, compressed_context):
    # Define a structured prompt template
    response_prompt = PromptTemplate(
        input_variables=["query", "context"],
        template="Use the compressed context below to answer the query
accurately.\n\nCompressed Context: {context}\n\nQuery: {query}\n\nAnswer:"
    )
    # Format the final prompt
    final_prompt = response_prompt.format(query=query,
context=compressed_context)
    # Generate response using LLM
    response = chat_model(final_prompt)

    return response.content

query = "What are the recent advancements in quantum computing?" #E
retrieved_docs = retriever.get_relevant_documents(query)
compressed_context = compress_context(retrieved_docs)
response = generate_response(query, compressed_context)

# Display Results
print("\n--- Retrieved Documents (Before Compression) ---")
for doc in retrieved_docs:
    print(f"Content: {doc.page_content}\n")

```

```
print("\n--- Compressed Context ---")
print(compressed_context)
```

```
print("\n--- LLM Response ---")
print(response)
```

#A Create sample documents with metadata and store them in a FAISS vector database

#B Initialize a ChatOpenAI model and define a retrieval pipeline for the top 3 relevant documents

#C Define a function to compress multiple retrieved documents into a concise summary

#D Define a function that uses the compressed context to generate a final answer

#E Demonstrate usage with a query, compress the retrieved context, and output the final answer

Let's look at the output of the program.

```
--- Retrieved Documents (Before Compression) ---
```

```
Content: Quantum computing research at MIT has led to breakthroughs in 2024,
including improved qubit coherence and error correction techniques.
```

```
Content: Recent advances in neuroscience have expanded our understanding of brain
function, particularly in memory storage and retrieval.
```

```
Content: The US stock market surged due to economic policy changes, leading to
record gains in the technology sector.
```

```
--- Compressed Context ---
```

```
MIT research in 2024 has led to advancements in quantum computing, improving
qubit stability and error correction. Neuroscience discoveries have enhanced
knowledge of memory function, while economic policy changes have driven stock
market gains.
```

```
--- LLM Response ---
```

```
Recent quantum computing breakthroughs at MIT in 2024 include improvements in
qubit coherence and error correction, advancing computational stability and
efficiency.
```

In Listing 6.14, we utilized FAISS using OpenAI embeddings for efficient vector-based retrieval. When a query is made, the system retrieves the top 3 relevant documents based on similarity. To improve response efficiency, the retrieved content is compressed using an LLM-based summarization chain, ensuring key facts are preserved while reducing redundancy. Finally, the compressed context is passed to GPT-4o to generate a well-informed response. This approach improves retrieval relevance, reduces unnecessary information, and enhances response quality for user queries. Next, we will examine re-ranking and how it helps RAG generate accurate responses.

RE-RANKING

In RAG systems, retrieving relevant chunks is the first step. Re-ranking refines retrieval by ensuring that the most contextually relevant information is prioritized. This is achieved using a cross-encoder model, which scores each retrieved chunk against the query with higher accuracy than the bi-encoders used in the initial retrieval stage. Additionally, it filters out irrelevant or weakly related information that could dilute response quality. Given the limited context windows of LLMs, re-ranking prioritizes the highest-ranked chunks, maximizing information density for more effective processing. Re-ranking functions by executing the following:

1. **Initial Retrieval (Bi-Encoder):** Utilizes a bi-encoder model (e.g., Sentence Transformers) to swiftly retrieve an initial set of relevant chunks from the database. This method is fast and scalable; however, its relevance assessment is limited since query and document embeddings are compared separately.
2. **Relevance Scoring (Cross-Encoder):** A cross-encoder model, such as MS MARCO or BERT-based models, evaluates the query and retrieved chunks to generate a more precise relevance score. In contrast to bi-encoders, which compare embeddings separately, cross-encoders process both the query and the text together, thereby enhancing semantic understanding and retrieval accuracy.
3. **Re-ranking and selection:** Retrieved chunks are sorted according to their relevance scores, and the top-K chunks are chosen for inclusion in the final prompt.

Now that we've learnt the re-ranking technique, let's implement it programmatically using LangChain.

Listing 6.15 shows the implementation of re-ranking using Lang Chain.

```
import nltk
from nltk.tokenize import word_tokenize
from rank_bm25 import BM25Okapi
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
from sentence_transformers import CrossEncoder
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate

nltk.download('punkt') # Download tokenizer resources
# Sample documents #A
documents = [
    Document(page_content="Quantum computing research at MIT has led to major
```



```

breakthroughs in 2024.", metadata={"category": "science", "date": "2024",
"organization": "MIT"}),
    Document(page_content="AI-powered diagnostic tools are revolutionizing
medical treatments.", metadata={"category": "healthcare", "date": "2023",
"organization": "Harvard Medical School"}),
    Document(page_content="Stock markets are influenced by inflation and global
economic shifts.", metadata={"category": "finance", "date": "2024",
"organization": "Federal Reserve"}),
    Document(page_content="Recent advances in neuroscience have expanded our
understanding of brain function.", metadata={"category": "science", "date":
"2023", "organization": "Stanford University"})
]
# Tokenize documents for BM25 Indexing
tokenized_corpus = [word_tokenize(doc.page_content.lower()) for doc in documents]
bm25 = BM25Okapi(tokenized_corpus) # Create BM25 index
# Initialize OpenAI embeddings
embedding_model = OpenAIEmbeddings()
# Store documents in FAISS for vector search
vector_db = FAISS.from_documents(documents, embedding_model)
# Load Cross-Encoder Model for Re-Ranking #B
cross_encoder = CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2")
# Function to retrieve documents using BM25 & FAISS #C
def retrieve_documents(query, top_n=3):
    # BM25 Retrieval
    tokenized_query = word_tokenize(query.lower()) # Tokenize query
    bm25_scores = bm25.get_scores(tokenized_query) # Compute BM25 scores
    bm25_ranked_indices = sorted(range(len(bm25_scores)), key=lambda i:
bm25_scores[i], reverse=True)[:top_n]
    bm25_results = [documents[i] for i in bm25_ranked_indices]
    # Dense Retrieval using FAISS
    retrieved_dense_docs = vector_db.similarity_search(query, k=top_n)
    # Combine retrieved documents
    combined_results = bm25_results + retrieved_dense_docs
    unique_docs = list({doc.page_content: doc for doc in
combined_results}.values()) # Remove duplicates
    return unique_docs
# Function to re-rank documents using a cross-encoder
def re_rank_documents(query, retrieved_docs):
    # Prepare query-document pairs for scoring
    query_doc_pairs = [(query, doc.page_content) for doc in retrieved_docs]
    # Compute relevance scores
    scores = cross_encoder.predict(query_doc_pairs)
    # Re-rank documents based on cross-encoder scores
    ranked_results = sorted(zip(retrieved_docs, scores), key=lambda x: x[1],

```

```

reverse=True)
    return [doc for doc, score in ranked_results]
# Function to generate response using re-ranked documents #D
def generate_response(query, retrieved_docs):
    chat_model = ChatOpenAI(model="gpt-4o", openai_api_key="
<<YOUR_OPENAI_API_KEY>>")
    # Define a structured prompt template
    prompt_template = PromptTemplate(
        input_variables=["query", "context"],
        template="Use the retrieved context below to answer the query
accurately.\n\nContext: {context}\n\nQuery: {query}\n\nAnswer:"
    )
    # Combine retrieved documents into context
    context = "\n".join([doc.page_content for doc in retrieved_docs])
    # Generate response using LLM
    final_prompt = prompt_template.format(query=query, context=context)
    response = chat_model(final_prompt)
    return response.content
query = "What are the latest advances in quantum computing?" #E
retrieved_docs = retrieve_documents(query, top_n=3)
re_ranked_docs = re_rank_documents(query, retrieved_docs)
response = generate_response(query, re_ranked_docs)
# Display Results
print("\n--- Retrieved Documents (After Re-Ranking) ---")
for doc in re_ranked_docs:
    print(f"Content: {doc.page_content}\n")
print("\n--- LLM Response ---")
print(response)

```

#A Define sample documents and create BM25 & FAISS indices

#B Load CrossEncoder model for re-ranking

#C Define functions for multi-stage retrieval

#D Define a function to generate an LLM-based response from re-ranked documents

#E Execute retrieval, re-ranking, and final response generation, then print results

The output of the program is as follows:

```
--- Retrieved Documents (After Re-Ranking) ---
```

```
Content: Quantum computing research at MIT has led to major breakthroughs in 2024.
```

```
Content: Recent advances in neuroscience have expanded our understanding of brain function.
```

```
--- LLM Response ---
```

```
Quantum computing has seen significant breakthroughs in 2024, particularly at MIT, where researchers have developed advanced qubit architectures improving stability and performance.
```

In Listing 6.15, we use a hybrid retrieval and ranking pipeline to address highly relevant user queries. It first utilizes the NLTK package for query tokenization and the BM25Okapi package to rank documents based on keyword relevance. Additionally, FAISS and OpenAI embeddings facilitate dense vector-based retrieval. The documents retrieved from both methods are then re-ranked using a Cross-Encoder model, which scores query-document relevance with greater accuracy. Finally, a structured ChatOpenAI prompt generates a well-grounded response based on the retrieved context, ensuring minimized hallucinations and improved answer quality.

GENERATION

Once the relevant documents are retrieved, they are passed to the generative model (like a transformer-based model) alongside the original query. The generative model produces an output, which could be a natural language response or answer. During this phase, the model leverages the retrieved documents to:

- **Condition the response:** The model enhances its generation by incorporating additional context from the retrieved information, thereby improving its ability to produce accurate and contextually relevant responses.
- **Refine its output:** The generative model can enhance its responses by utilizing the details from the retrieved documents, allowing it to capture specific nuances that may not appear in the original input query.

By now, you should have a good understanding of an RAG system, how it works, the components of an RAG, and its architecture. With the help of this chapter, you should be able to create your own mini-RAG system for your specific use case. The next chapter will review the different types of RAG systems, how to implement them for your use cases, and their pros and cons.

6.6 Summary

- We learned about Retrieval-Augmented Generation and its importance in reducing hallucinations and improving the accuracy and relevancy of user prompt responses. We also learned about the limitations of generic models trained on very vast datasets.
- RAG addresses these limitations of traditional language models by integrating a retrieval step into the response-generation process. This approach ensures that the responses are accurate and backed by relevant information, reduced hallucination, and the model can leverage on-demand readily available knowledge to respond to queries.
- Prompt engineering plays a pivotal role in the RAG system. Prompt patterns and templates are widely used in the Post-Retrieval phase before the model generates a response.
- A mini-RAG workflow can be seen leveraging models generally available by using prompts and uploading documents or data relevant to the specific use case. However, for large-scale usage, such as in a company or a more specific environment, a RAG implementation helps reduce hallucination, generates accurate and relevant responses, and saves time in tuning the model, as RAG is significantly cheaper to implement.
- The RAG workflow consists of Pre-Retrieval, Retrieval, and Post-Retrieval phases. The Pre-Retrieval phase consists of data collection, indexing, chunking, embedding generation, and storage in vector databases. This phase is done before the RAG system is ready for users to use for their prompts/queries.
- We learned about sparse, dense, hybrid retrievers and their use cases. The sparse retriever is based on lexical matching, the dense retriever on semantic matching of words, and the hybrid retriever combines both sparse and dense approaches.
- The retrieval phase involves real-time processing of the user's prompt. In this phase, the RAG system parses the prompt, refines it to address any vagueness, and retrieves relevant data.
- The post-retrieval phase is the final stage before the model generates a response. Once the data is obtained, the prompt and the retrieved data are forwarded to the model for processing. The prompt provided to the model follows a standardized template for responses and inquiries. This approach standardizes outputs and ensures specific patterns for various use cases. Before forwarding, prompt compression is utilized to accommodate the model's limitations on the number of tokens it can process. Re-ranking is also performed to send only the most relevant data while ignoring other retrieved context.

7 Types of RAG Systems: A Deep Dive

This chapter covers

- Naïve RAG, Graph RAG, Hybrid RAG, and Memory Augmented RAG, their components, workflow, and use cases.
- For each RAG system, we will examine a practical example using sample data to help us understand how RAG systems operate.
- We will see how each RAG differs and when to use them.
- A comparison table of each RAG system that helps you choose the appropriate RAG implementation for your use case.

Chapter 6 discussed RAG in detail, focusing on its components. We thoroughly examined how a RAG system operates by incorporating external data sources into the model's query processing. We explored the benefits, architectural review, types of retrievers, advantages, and use cases of RAG components.

In this Chapter, we will examine different types of RAG systems based on the architecture, advantages, and stages discussed in Chapter 6. We will examine each RAG system using one real-world example. This example will help you understand how to implement a RAG system for custom use cases. After discussing each RAG system, we will also look at its pros and cons, and the example with its results will help you understand which use cases you can solve using the different RAG systems.

There are many RAG systems, with the most basic one called the Naïve RAG system. At its core, each RAG system has the same or similar components that we discussed in Chapter 6. The difference is how each RAG system pre-processes the query and external data source, processes the query and context, and post-processes the response generated by the LLM. Let's dive deep into each one of them with real world examples.

7.1 Naïve RAG

A naive RAG system is the most basic implementation of RAG, designed to enhance language model responses by retrieving relevant documents from an external knowledge base. It is termed "naive" because it follows a simple retrieval-then-generation pipeline without sophisticated optimizations, such as query reformulation, multi-hop reasoning, or advanced reranking mechanisms. While effective for many use cases, it often faces challenges like retrieving irrelevant context, a lack of dynamic adaptation, and redundancy in retrieved information. The fundamental architecture of a Naïve RAG system includes two primary components: retrieval and generation. The retrieval component typically utilizes a vector database, such as FAISS or Pinecone, where documents are stored as embeddings for efficient similarity search. When a user submits a query, the system transforms it into an embedding and retrieves the top-K most relevant documents based on cosine similarity or another distance metric. These documents provide additional context for the generative model.

In Figure 7.1, we see the simplistic nature of Naïve RAG system. A Naive RAG setup transforms a user query into an embedding to retrieve relevant documents from a vector store or search index. These retrieved chunks are then concatenated and passed to the LLM as part of the prompt context, enabling the model to generate grounded responses that reflect the most relevant external data.

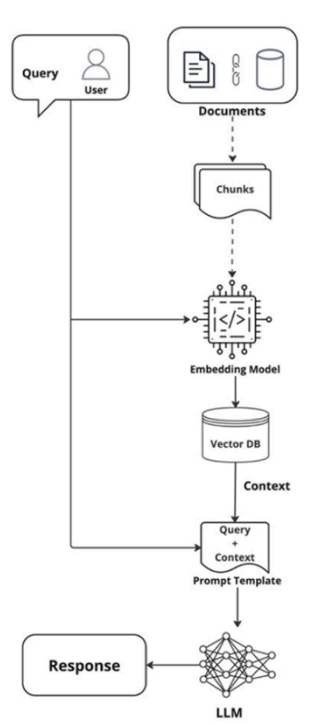


Figure 7.1 shows a Naïve RAG system.

Naive Retrieval-Augmented Generation (RAG) is the most straightforward and most accessible method for connecting a language model (LLM) with external knowledge. It allows developers to “bolt on” a document corpus to an LLM without requiring model fine-tuning or complicated retraining. In this design, the model retrieves a few relevant documents based on a user input and uses them as context to generate its response. No dynamic memory, no multi-hop logic — just fast, targeted lookup plus generation. Despite its simplicity, Naïve RAG delivers strong performance for use cases such as FAQs, static corpora (e.g., manuals, product documentation), and chatbot scenarios. It is cheap to build, easy to scale, and serves as a clean starting point for more advanced RAG architectures. For many projects, Naïve RAG is “good enough.” This section walks through how Naïve RAG works — from vector indexing to generation — and shows how you can build it from scratch using real-world data and open-source tools.

A well-known user-facing example of Naïve RAG is Google’s NotebookLM, which lets users upload source documents and then chat with an LLM that answers based on those materials. Behind the scenes, it uses vector search to retrieve relevant text chunks and passes them to the model — exactly the workflow described in this section. Other tools like ChatGPT with “custom instructions + file uploads” or document-focused apps like Perplexity’s “Library” feature follow similar designs.

This straightforward pipeline offers a compelling baseline for knowledge-augmented generation without requiring model fine-tuning or complex orchestration. Naïve RAG systems are beneficial in scenarios where speed of implementation, ease of integration, and low operational complexity are essential.

PROMPTING

Naïve RAG relies entirely on the LLM to make sense of the flat retrieved context. Prompt patterns like “answer based on the context below” or “only use the provided documents” are essential to avoid hallucinations. Persona prompts or explicit grounding instructions can also help constrain generation.

Example: “Answer the following question using only the information from the retrieved articles. Be specific and cite document snippets if needed.”

Let’s walk through at a detailed workflow of a Naïve RAG system and see what each component does.

7.1.1 Core Components of a Naïve RAG System

Figure 7.2 shows the core components of a Naïve RAG system via a flowchart.

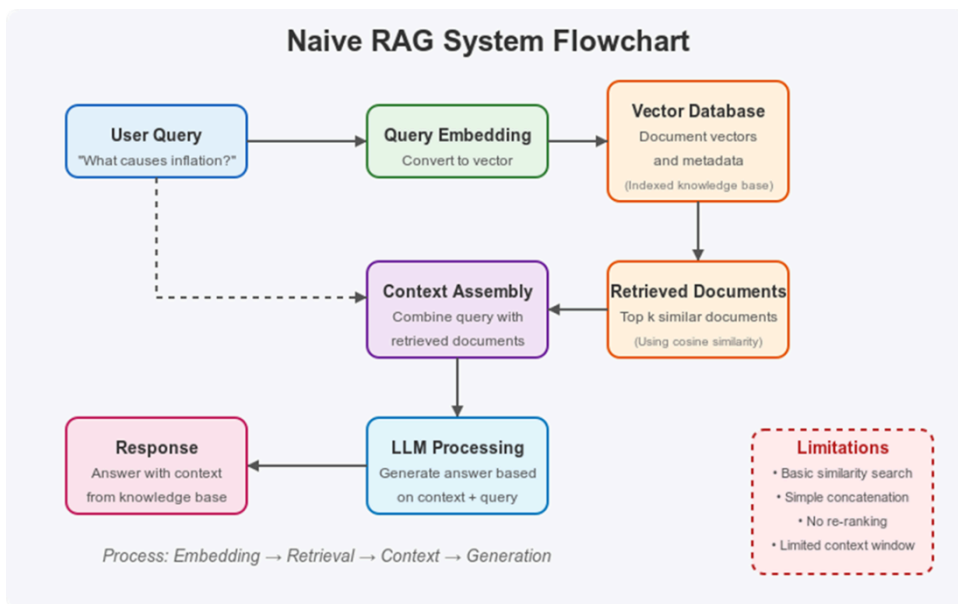


Figure 7.2 A flowchart of a Naïve RAG system.

Let’s walk through each step and explain how it works in detail. Figure 7.2 assumes that the documents have been chunked, embeddings created, and stored in a vector database.

DOCUMENT COLLECTION AND KNOWLEDGE BASE

The foundation of any RAG system is its knowledge base: a collection of documents containing the information the system can access when answering queries. In a naive implementation, these documents can include any text-based resource: web pages, PDFs, technical manuals, scholarly articles, or internal company documentation. Documents are typically divided into smaller, manageable chunks (paragraphs or sections) that can be processed efficiently. While the raw documents may be stored in their original form, the system requires efficient retrieval mechanisms, which are typically implemented as a vector database. This collection of documents provides the contextual information that the LLM wouldn't otherwise have access to, allowing it to answer questions based on specific information rather than just general knowledge. In Figure 7.3, this is shown as the external documents/data source being chunked, embeddings generated, and then stored in a vector database.

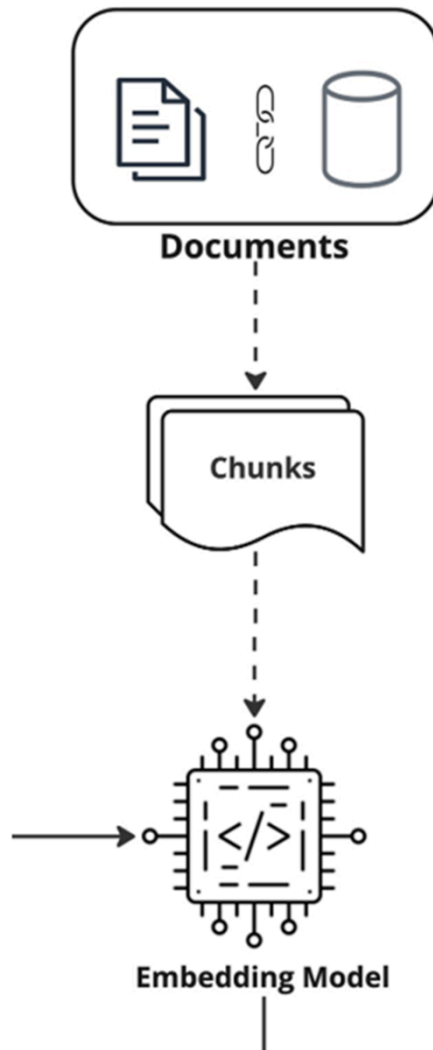


Figure 7.3 The document processing part of the Naïve RAG system.

RAG systems typically split documents into chunks to enable more precise retrieval. But not all chunk sizes are created equal:

- Sentence-level chunks offer high granularity and work well when questions are particular (e.g., fact lookup, short definitions).
 - Used in: Chatbots, compliance queries.
- Paragraph-level chunks preserve more local context and are suitable for summarization or questions about flow, tone, or logic.
 - Used in: Document Q&A, product manuals.

- Section-level chunks are larger units and work well when full context is needed to avoid misinterpretation.
 - Used in: Legal, finance, and academic analysis.

Choosing the right granularity depends on the task and how “narrow” or “broad” your typical queries are.

The embedding model transforms text into numerical vector representations that capture semantic meaning. This is crucial for both query and document processing. Document embedding involves converting each chunk into a high-dimensional vector (typically 768-1536 dimensions) representing its semantic content. User queries undergo the same transformation process to enable similarity comparisons. Typical embedding models include those from the sentence-transformers family, OpenAI's embedding models, and other specialized text embedding models. Basic text normalization may be applied before embedding generation, such as lowercasing, removing special characters, or basic tokenization. The quality of the embedding model significantly impacts retrieval performance, as it determines how effectively semantic similarity can be measured.

USER QUERY PROCESSING

In Figure 7.2, we use a sample query, “What causes inflation?” as the user's input. The query, like the documents, goes through the embedding model. User queries undergo an embedding transformation process to enable similarity comparisons. Once the embeddings are generated for the user query, the retriever searches the vector database to find relevant document chunks based on query similarity. It uses vector comparison techniques (typically cosine similarity) to identify document chunks closest to the query vector. In naive implementations, the retriever selects a predetermined number (k) of the most similar documents, typically 3-5. It may apply basic metadata filters, but generally lacks sophisticated filtering in naive implementations.

CONTEXT ASSEMBLY

Once the relevant documents are retrieved, they must be combined with the original query to form a prompt for the LLM. In naive RAG, retrieved document chunks are simply concatenated together, often with separators to delineate different chunks. A template combines the user query with retrieved documents, often including instructions for how the LLM should use the context. The system must ensure that the combined prompt fits within the LLM's context window limitations. In naive implementations, documents are typically organized by similarity score without sophisticated ordering strategies. This step is called the “stuff” approach in RAG terminology.

RESPONSE GENERATION

The LLM is the generator component that produces coherent responses based on the user query and retrieved context. It reads the query and the retrieved contexts, identifies relevant information, and synthesizes a response. It must differentiate between general knowledge and specific information from retrieved documents. The model generates natural language responses that address the user's query while incorporating relevant retrieved information. In naive implementations, citation or attribution capabilities are typically limited or absent. The generator's effectiveness depends on the retrieved contexts and the LLM's ability to integrate this information coherently.

The Naïve RAG pipeline we'll build follows the standard architecture described in Figure 7.2: we start by loading and chunking a document corpus (step 1), embedding the chunks using a pre-trained model (step 2), and storing the embeddings in a vector database for similarity search (step 3). Once that's in place, we process a user query by generating its embedding (step 4), retrieving the top-matching chunks, and feeding them — along with the original query — to a language model to generate the final answer (step 5). Each of the steps below maps directly to this flow.

7.1.2 Building a Naïve RAG system from scratch

Now that we understand how a Naïve RAG system works, let's build a Naïve RAG system from scratch and see how it works. Listing 7.1 shows the first two steps of a Naïve RAG system implementation.

Listing 7.1 shows step 1 & 2 of a Naïve RAG implementation

```
import required libraries
import nltk
from nltk.corpus import reuters
from sentence_transformers import SentenceTransformer
import chromadb
from chromadb.config import Settings
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv
import os
from langchain import PromptTemplate, LLMChain
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage
os.environ["TOKENIZERS_PARALLELISM"] = "false"
load_dotenv()
# Download the Reuters corpus if not already available
if not reuters.fileids():
    nltk.download('reuters')
# -----
# Step 1: Load Corpus and chunk documents
```

```

# -----
# Load and use the first 500 Reuters documents
fileids = reuters.fileids()
docs = [reuters.raw(fileid) for fileid in fileids[:500]]          #A
print(f"*****")
print(f"Step 1: Using {len(docs)} Reuters documents for processing.")
print(f"*****")
# -----
# Step 2: Embedding Generation with Sentence-Transformers
# -----
print(f"*****")
print("Step 2: Generating embeddings...")
print(f"*****")
model = SentenceTransformer('all-MiniLM-L6-v2')                  #B
embeddings = model.encode(docs, show_progress_bar=True)

```

#A Load data from Reuters for analysis.

#B Generate embeddings using the MiniLM-L6 sentence transformer.

Listing 7.1 shows the first two steps in building the Naïve RAG system. As we progress, you will notice the initial steps: importing libraries, loading the required data, and generating embeddings. In the Naïve RAG implementation, we utilize the `all-MiniLM-L6-v2` sentence transformer. The `all-MiniLM-L6-v2` model is a compact and efficient sentence transformer developed by the SentenceTransformers team. It is based on the MiniLM architecture, a distilled variant of BERT, designed to maintain strong performance while dramatically reducing model size and computational load. With only 33 million parameters, producing 384-dimensional embeddings, this model balances speed, memory usage, and semantic understanding. It was trained using contrastive learning on a large collection of sentence pairs to effectively capture semantic similarity, making it suitable for a wide range of natural language understanding tasks. The model is widely available through Hugging Face and integrates easily with standard Python frameworks.

One of the key advantages of `all-MiniLM-L6-v2` is its inference speed. Its small size makes it ideal for low-latency environments, even on CPUs or smaller GPUs such as the T4 or RTX series. Additionally, because the embeddings have a lower dimensionality compared to larger models (384 vs. 768 or more), vector storage and similarity searches are faster and more memory efficient. Despite its compact size, it provides surprisingly strong performance for semantic search and retrieval, handling paraphrasing and meaning-level matching well. It is also highly accessible, featuring a simple plug-and-play interface via the sentence-transformers Python library, and works seamlessly with tools like FAISS, LangChain, or Haystack.

This model is commonly used in various scenarios where semantic matching is important, but resources are limited. It is particularly effective in naive RAG (Retrieval-Augmented Generation) pipelines, where queries and documents are embedded in the same vector space to retrieve relevant chunks of information before generating a final answer using a language model. Beyond RAG, it is also useful for building semantic search engines, powering FAQ bots, searching knowledge bases, or clustering documents by meaning. Thanks to its efficiency and versatility, `all-MiniLM-L6-v2` has become a go-to choice for developers and researchers seeking to build performant NLP applications without the overhead of larger transformer models.

What is a vector database? Once embeddings are generated, we need a way to store them so that we can later retrieve the most relevant chunks for any user query; this is where a vector database comes in. Unlike traditional databases, which search text or structured fields, vector databases perform fast similarity searches over high-dimensional embeddings. They return the “nearest neighbors” — i.e., chunks most semantically similar to the query vector. **What is ChromaDB?** ChromaDB is a lightweight, open-source vector database optimized for local development and prototyping. It stores document embeddings and allows fast approximate nearest-neighbor (ANN) queries based on cosine similarity or other metrics. In this example, we use ChromaDB to store and search over the embedded Reuters document chunks.

Next, in Listing 7.2, we will see steps 3 and 4 of building the Naïve RAG system.

Listing 7.2 shows steps 3 & 4 of the Naïve RAG system

```
# -----
# Step 3: Setting up ChromaDB for Naive RAG
# Now that we have embeddings, we create a ChromaDB client and store them
# -----
print(f"*****")
print("Step 3: Setting up ChromaDB...")
print(f"*****")
# Create a ChromaDB client; this example runs in-memory. #A
# Ensure only one instance of ChromaDB is created
try:
    client = chromadb.Client(Settings(
        chroma_db_impl="duckdb+parquet",
        persist_directory=".chromadb_data"
    ))
except ValueError as e:
    if "An instance of Chroma already exists" in str(e):
        print("Reusing the existing ChromaDB instance.")
        # Optionally, handle reusing the existing instance if needed
    else:
        raise e
# Create or get a collection to store Reuters embeddings and corresponding
```

```

documents.
collection = client.get_or_create_collection(name="naive_rag_reuters")
# Prepare data for indexing (IDs, documents, embeddings) #B
ids = [str(i) for i in range(len(docs))]
documents = docs
embeddings_list = embeddings.tolist()
# Add documents to the ChromaDB collection.
collection.add(
    ids=ids,
    documents=documents,
    embeddings=embeddings_list
)
print("Documents indexed in ChromaDB.") #C
# -----
# Step 4: Querying ChromaDB - Example Search
# -----
print("*****")
query_text = "Analyze how oil price fluctuations have influenced stock markets,
currency valuations, and international trade over the past decade. Highlight the
causal relationships among these factors. Can you also highlight the points to
support the response?"
print(f"Step 4: Running query: '{query_text}'")
print("*****")
query_embedding = model.encode([query_text]).tolist()[0]
# Search the collection; here we get the top 3 most similar documents.
query_results = collection.query(
    query_embeddings=[query_embedding],
    n_results=3,
    include=["documents", "distances"]
)
# Parse the results and prepare a context snippet
retrieved_info = []
for idx, doc in enumerate(query_results["documents"][0]):
    retrieved_info.append({
        "id": query_results["ids"][0][idx],
        "distance": query_results["distances"][0][idx],
        "text": doc
    }) #D

```

#A Set up ChromaDB for storing embeddings, documents, etc.

#B Iterate through the embeddings, documents and add them to the database.

#C Print documents indexed in the database.

#D Run a sample query to check the data fetched from the database.

Listing 7.2 illustrates how we set up ChromaDB for the Naïve RAG system. We chose ChromaDB because it provides a fast and straightforward setup, allowing you to get started quickly without complex infrastructure. It natively supports storing embeddings and metadata, eliminating the need to connect separate vector stores and databases. ChromaDB is optimized for rapid local retrieval using similarity search methods like cosine distance, making it ideal for quick document lookups in RAG systems. Additionally, it automatically persists data, ensuring that embeddings survive restarts without requiring re-indexing. Its lightweight nature means you don't need a separate server or cluster, which perfectly suits small-to-medium datasets typically used in early-stage RAG projects. We, then, run a sample query to check the documents fetched based on embeddings and similarity search.

In this final step, we combine the retrieved document chunks with the original query and generate a response using a large language model. We use GPT-4o from OpenAI for generation, while the earlier embedding steps relied on all-MiniLM-L6-v2, a lightweight sentence transformer used solely for semantic search. These models serve different roles: MiniLM for retrieval, GPT-4o for response generation. Next, we will see the final step of the RAG system, where we generate a response. Listing 7.3 shows the final step.

Listing 7.3 shows step 5 of building the Naïve RAG system

```
# -----
# Step 5: Generate Final Response Using GPT
# -----
def generate_summary_with_gpt(query_text, retrieved_info):          #A
    context_snippets = "\n\n".join([
        f"Document ID {info['id']}: \n{info['text'][:300]}..." for info in
retrieved_info
    ])
    prompt = f"""
    We have retrieved a few documents based on the query: "{query_text}".
    Below are summaries extracted from the top matching documents:
    {context_snippets}
    Please provide a clear, concise summary that combines the key insights about
current financial market trends and investment strategies.          #B
    """
    print("Generating final response using ChatGPT API...")
    try:
        response = openai.chat.completions.create(
            model="gpt-4o",
            messages=[
                {"role": "system", "content": "You are a financial assistant that
provides concise, insightful answers."},
                {"role": "user", "content": prompt}
            ],
```



```

        temperature=0.7,
        max_tokens=300
    )
    answer = response.choices[0].message.content
    print("Response generated successfully.")
    print(f"AI: {answer}")
except Exception as e:
    answer = f"OpenAI API Error: {e}"
    # Output result
    print(f"\nUser: {query}")
    print(f"AI: {answer}")
    print(f"Sources: {sources}")
# -----
# Execution of Naive RAG
# -----
def main():
    docs = load_reuters_docs()
    model, embeddings = generate_embeddings(docs)
    collection = setup_chromadb(docs, embeddings)
    static_query = "Analyze how oil price fluctuations have influenced stock
markets, currency valuations, and international trade over the past decade.
Highlight the causal relationships among these factors. Can you also highlight
the points to support the response?"
    retrieved_info = retrieve_documents(collection, static_query, model)
    generate_summary_with_gpt(static_query, retrieved_info)
if __name__ == "__main__":
    main()

```

#A Create context for response generation. In this step, we also retrieve the top 3 relevant documents.

#B Create the prompt that we will send to ChatGPT along with the context.

#C Set the max tokens, use the persona prompt pattern, and prompt the model with the context to generate the response.

#D Load the documents, generate embeddings and send a query to the Naïve RAG system.

Listing 7.3 shows the final step of the Naïve RAG system. In this step, we retrieve related documents from the database by processing the prompt/query. Next, we use ChatGPT, assigning the model the persona of an expert financial summarizer to generate the final response. Listing 7.4 presents the final response from the RAG system.

Listing 7.4 shows the final response from a Naïve RAG system.

Using 500 Reuters documents for processing.

Generating embeddings...

Batches: 100%|

16/16 [00:01<00:00, 10.03it/s]

Setting up ChromaDB...

Documents indexed in ChromaDB.

```
Running query: 'Analyze how oil price fluctuations have influenced stock markets,
currency valuations, and international trade over the past decade. Highlight the
causal relationships among these factors. Can you also highlight the points to
support the response?'
```

Top retrieved documents:

ID: 33 - Distance: 1.1248

ID: 272 - Distance: 1.2208

ID: 387 - Distance: 1.2336

Generating final response using ChatGPT API...

Response generated successfully.

AI: Based on the document summaries provided, we can draw some insights about the influence of oil prices on financial markets and related economic factors:

1. **Oil Prices and Economic Recovery**: The Kuwaiti economy, heavily reliant on oil, demonstrates how fluctuations in oil prices can significantly impact national economic recovery and stability. This relationship underscores the critical role of oil prices in determining the pace of economic growth, particularly for oil-exporting countries.

2. **Demand and Supply Dynamics**: The decrease in U.S. oil demand, as reported, indicates broader trends in energy consumption that can affect global oil prices. Changes in demand can lead to fluctuations in oil prices, which in turn can influence stock markets and currency valuations as economies adjust to new levels of energy costs.

3. ****Commodity Prices and Market Balance****: The rise in lead prices due to supply-demand dynamics highlights how commodity markets can be affected by external factors such as geopolitical events or labor issues. These changes can also influence broader financial markets as investors adjust their strategies based on perceived risks and opportunities.

****Causal Relationships**:**

- **Oil Prices and Stock Markets**: Fluctuations in oil prices can lead to volatility in stock markets, particularly for companies in energy-dependent sectors. High oil prices can increase costs for businesses, affecting

profitability and stock valuations.

```
- **Oil Prices and Currency Valuations**: Changes in oil prices can influence
currency values, especially for countries that are major importers or exporters
of oil. An increase in oil prices
```

Listing 7.4 illustrates the query sent to the RAG system and the response received. Analyzing the response, we see it contains information on how oil prices affect the overall economy. The response also includes points that support the data presented. It connects fluctuations in oil prices to an increase in inflation, which leads to stock market volatility, among other effects. This response on its own could be a good response. We will revisit this response as we learn about other RAG techniques. Next, let's explore the pros, cons, and use cases of a Naïve RAG system.

7.1.3 Pros, Cons, and Use Cases

PROS

From a development and deployment standpoint, naive RAG systems are simple, modular, and cost-effective. They can be constructed using widely available tools, like FAISS or OpenSearch for retrieval and commercial LLM APIs for generation, and require minimal configuration. This makes them an ideal architecture for prototyping AI assistants, testing information access patterns, or validating user experience flows before committing to more advanced pipelines. For many teams, especially in early-stage or low-risk environments, naive RAG represents a strong starting point for bringing retrieval-augmented intelligence to real-world applications.

CONS

However, naive RAG systems have notable limitations that constrain their performance and scalability despite their utility. A primary concern is the loose coupling between retrieval and generation. The retrieval step typically uses vector similarity to identify top-k documents, but this selection occurs without deeper reasoning or iterative refinement. Consequently, the retrieved chunks often vary in relevance, and irrelevant or redundant information can pollute the prompt context. Since the LLM receives all retrieved content in a flat structure, often hitting token limits, it may fail to prioritize or even attend to the most critical information, leading to generic, inconsistent, or hallucinated outputs.

The lack of feedback loops or dynamic control mechanisms means that naive RAG cannot adapt retrieval based on generation progress. It also cannot reason across documents, verify factual claims, or handle multi-hop queries that require chaining multiple pieces of information. This limitation renders naive RAG insufficient for high-stakes domains such as legal, medical, or financial analysis, where output reliability and factual correctness are critical. Additionally, evaluating such systems is difficult: it's often unclear whether a poor response arises from faulty retrieval or model generation, and the lack of transparency hampers efforts to debug or enhance performance.

USE CASES

Naive RAG systems are beneficial in scenarios where speed of implementation, ease of integration, and low operational complexity are important. One of their key strengths lies in customer support applications, particularly for answering FAQs or navigating internal knowledge bases. In such contexts, the corpus tends to be relatively static, well-structured, and small enough that top-k retrieval generally surfaces high-signal content. This makes naive RAG a practical solution for building virtual assistants that respond to user queries in a personalized yet reliable manner, reducing the load on human support agents and improving response consistency.

They also excel in enterprise search and knowledge discovery tools, where employees must find relevant information across internal documentation, wikis, or procedural manuals. With naive RAG, users can issue natural language queries and receive responses grounded in actual organizational knowledge, eliminating the need to sift through search result pages. Educational and training tools benefit similarly: by embedding textbooks, guides, or curriculum materials, naive RAG can generate explanations, summaries, or practice questions tailored to the learner's query. In creative and exploratory use cases like content generation, literature reviews, or brainstorming sessions, naive RAG provides a low-friction mechanism to synthesize information from multiple documents.

In summary, naive RAG systems provide an accessible and robust baseline for retrieval-augmented applications. They are particularly effective in well-scoped, low-risk domains where static corpora and simple queries prevail, and they offer a quick path to enhance LLM capabilities with external knowledge. However, for more complex tasks that require contextual reasoning, precise factual grounding, or iterative refinement, more sophisticated RAG architectures that incorporate reranking, multi-hop retrieval, or retriever-generator co-training are essential to overcome the inherent limitations of the naive approach. The following is a quick-reference summary of the pros, and cons described above.

PROS

- Speed of implementation - Quick to set up using widely available tools
- Ease of integration - Works with commercial LLM APIs and common retrieval systems like FAISS or OpenSearch
- Low operational complexity - Minimal configuration required
- Cost-effective - Utilizes existing tools with minimal overhead
- Ideal for prototyping - Good starting point before committing to advanced pipelines
- Effective for static, well-structured corpora - Works well when information changes infrequently
- Strong for FAQs and knowledge bases - Performs well with straightforward lookups
- Reduces human support workload - Automates common information retrieval tasks
- Improves response consistency - Provides standardized answers to common queries

- Enables natural language enterprise search - Users can query in plain language
- Supports educational applications - Can generate explanations and summaries

CONS

- Loose coupling between retrieval and generation - No deeper reasoning or iterative refinement in document selection.
- Inconsistent relevance - Retrieved chunks often vary in relevance quality.
- Context pollution - Irrelevant or redundant information can contaminate the prompt context.
- Token limit constraints - Retrieved content often hits model context limits.
- Poor prioritization - LLM may fail to focus on the most critical information.
- No feedback loops - Cannot adapt retrieval based on generation progress.
- Limited reasoning across documents - Cannot effectively chain information for multi-hop queries.
- Verification challenges - Cannot verify factual claims independently.
- Insufficient for high-stakes domains - Not reliable enough for legal, medical, or financial analysis.
- Difficult evaluation - Hard to determine if errors stem from retrieval or generation.
- Limited transparency - Hampers debugging and performance enhancement efforts

7.2 Graph RAG

Graph RAG represents an advanced method for enhancing Retrieval Augmented Generation (RAG) by integrating structured knowledge graphs into the retrieval and generation process. Rather than relying on simplistic, plain text-based semantic searches, Graph RAG establishes a hierarchical framework that organizes information more effectively. The process begins by extracting a knowledge graph from raw text, identifying key entities and their interrelationships.

Next, the method builds a community hierarchy where related entities and concepts are grouped into coherent clusters. For each cluster, tailored summaries are generated that encapsulate the essential information and contextual connections. These structured summaries inform the RAG process during both the indexing and querying phases, resulting in contextually richer responses that are more precise and better grounded in the underlying data. Let's look at Figure 7.4, which explains how a Graph RAG system works.

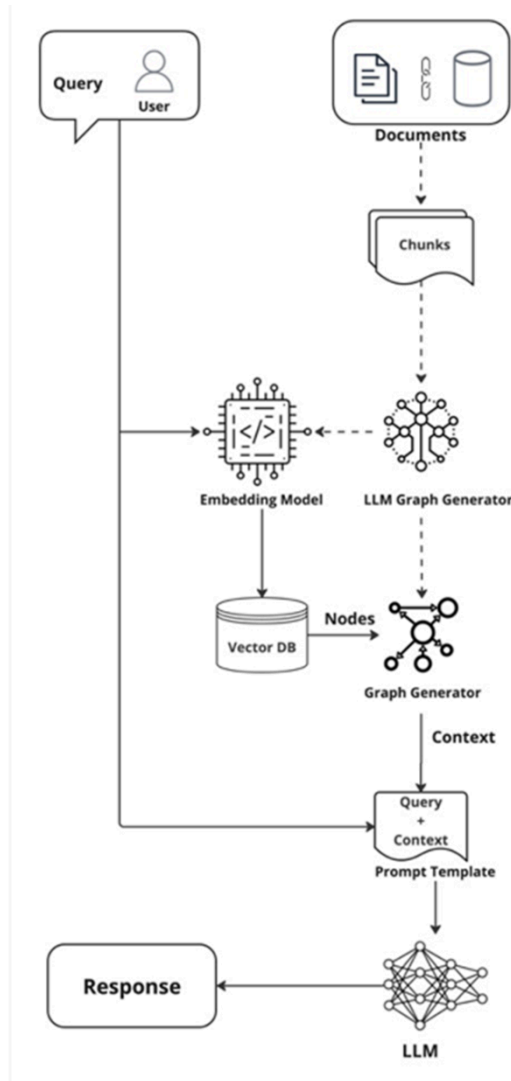


Figure 7.4 shows how a Graph RAG system works.

Let's look at the components of a Graph RAG system and see in detail how each component works.

7.2.1 Components of a Graph RAG system

Like the Naïve RAG system, the Graph RAG system has the same base components. The difference is around how these components work and interact with other components in the system. Let's look at each one of them to understand how they work. After that, we will take a closer look at the overall workflow.

DOCUMENT COLLECTION AND KNOWLEDGE BASE

The document collection component serves as the foundation of any RAG system. However, a Graph RAG system has a more rigorous indexing step. In Figure 7.3, this is illustrated by the workflow from documents to chunks to the LLM graph generator and the vector DB storage. The indexing steps form the backbone of the Graph RAG indexing process, transforming raw text into a richly annotated and hierarchically organized knowledge base that can be effectively leveraged in downstream RAG applications. The steps involved in indexing are as follows:

- **Text Unit Segmentation:** The process begins by dividing the entire input corpus into smaller, logical text units, such as paragraphs, sentences, or other well-defined segments. This segmentation ensures that even lengthy documents are broken down into manageable chunks, allowing a more granular extraction of details.
- **Entity, Relationship, and Claims Extraction:** In each segmented text unit, large language models (LLMs) are used to identify and extract critical components from the text. This includes names of people, places, and organizations (entities), as well as the relationships between these entities and key claims made within the text. The extracted information forms the foundation of an initial knowledge graph.
- **Hierarchical Clustering** Once the initial knowledge graph is constructed, the Graph RAG system applies hierarchical clustering using the Leiden algorithm—a robust community detection technique. This method effectively detects and organizes the graph into distinct communities, grouping together related entities and relationships. The clustering process helps reveal the data’s inherent structure and thematic connections.
- **Community-Level Summarization** Finally, each identified community undergoes summarization. These structured summaries encapsulate the essential information and insights of each cluster, providing a concise yet comprehensive overview. Such summaries are critical for later stages, as they enrich the context used during the querying phase and enhance the overall accuracy and detail of the RAG-based tasks.

Next, let’s look at the retrieval component of the Graph RAG system.

RETRIEVAL AND QUERY PROCESSING

Graph RAG supports two distinct querying workflows, each designed to address different types of questions:

Local Search

The local search workflow is designed to manage queries focused on particular entities within a knowledge graph. It excels at delivering precise, context-rich responses by exploring the immediate structure surrounding an entity—such as its neighbors, related concepts, and contextual attributes. When users submit questions centered around a specific entity, this workflow is recommended because it can delve into the fine-grained, local relationships that define the entity's role in the broader knowledge system.

The process begins with the reception of user queries, where the system captures the user's input, ranging from straightforward factual questions to more nuanced or exploratory queries. Following this, the system initiates a similar entity search to identify a set of semantically related entities within the knowledge graph. This is typically achieved through vector similarity searches using tools like Milvus, which serve as entry points into the graph's structure by surfacing closely related nodes.

Once relevant entities are identified, the next step involves mapping entity-text units. In this phase, extracted text segments are linked to their corresponding entities while filtering out irrelevant or noisy content. This ensures that only the most pertinent information is retained for downstream processing. The workflow then continues to entity-relationship extraction, where the system identifies and captures the relationships between entities, enriching the core context of the entity in question and building a more nuanced representation of its connections.

The system performs entity-covariate mapping to provide even deeper insights, linking entities to additional attributes or statistical data that can offer supporting quantitative or contextual information. These covariates deepen the system's understanding and enhance the final response. Further contextualization is achieved through entity-community report mapping, which incorporates global insights derived from community-level reports. These reports are sourced from higher levels of the community hierarchy and integrated into the entity-specific data to balance local precision with a broader perspective.

If the user has an active or ongoing interaction with the system, the conversation history is utilized to refine the query interpretation and tailor the response more accurately. This use of conversational context allows the system to disambiguate intent, adapt to evolving questions, and ensure consistency across multiple exchanges. Finally, all filtered and enriched data is consolidated in the response generation phase. Here, the system assembles a well-organized, context-aware reply that reflects both local details and global relevance, ensuring that the user receives a highly tailored and informative answer.

By integrating multiple layers of entity-specific analysis and global context, the local search workflow provides a powerful and methodical approach to information retrieval within knowledge graph systems. Its structured nature enables it to deliver responses that are not only precise but also aligned with the user's intent and informational needs.

Global Search

The global search workflow is optimized for holistic queries that demand a broad, integrated understanding of the entire data corpus. Rather than focusing on a specific entity, this approach leverages the structural overview provided by precomputed community summaries, enabling the system to reason across overarching themes and cross-cutting patterns in the data. This comprehensive perspective is particularly valuable when users pose questions requiring the synthesis of diverse sources or the interpretation of macro-level insights. By incorporating hierarchically organized knowledge from across the graph, global search facilitates rich, thematic analysis that contextualizes local information within a wider frame.

The process begins with user input integration, where the system assimilates the user's current query along with any available conversation history. This combined context serves as the foundation for downstream reasoning, ensuring continuity and relevance across multi-turn interactions. Next, the system retrieves community report batches, which are language model-generated summaries extracted from specific levels of the knowledge graph's community hierarchy. These reports are shuffled and segmented into multiple logical subsets (e.g., Batch 1, Batch 2), enabling distributed and parallel processing of contextual information.

Each batch of community reports is then broken down into smaller, predefined-sized text chunks, which are processed individually to produce Rated Intermediate Responses (RIRs). These responses contain concise points derived from the source material, each annotated with a numerical score reflecting its estimated importance or relevance. The scoring system aids in downstream filtering and ranking of information, ensuring that the most valuable insights rise to the top.

Following the generation of RIRs, the system moves on to a ranking and filtering phase. Individual points across all intermediate responses are sorted and curated during this phase based on their associated scores. Only the most informative and contextually relevant insights are retained, forming a refined collection called Aggregated Intermediate Responses. These aggregated insights serve as a distilled and high-signal representation of the broader knowledge base.

Finally, in the response generation phase, the system draws on these aggregated intermediate responses to compose a comprehensive reply to the user's query. This reply integrates information from multiple perspectives and layers of the knowledge graph, delivering a synthesized and well-rounded answer. By leveraging structural hierarchy and model-assisted summarization, the global search workflow enables rich, context-aware generation that is particularly well-suited for complex, open-ended questions that transcend individual data points or entities.

Let's look at the workflow of a Graph RAG system and understand how it works.

7.2.2 Workflow of a Graph RAG system

Figure 7.5 shows the workflow of a Graph RAG system. At a high level, the Graph RAG system enhances traditional RAG by incorporating knowledge graph structures alongside vector embeddings. The workflow operates as follows:

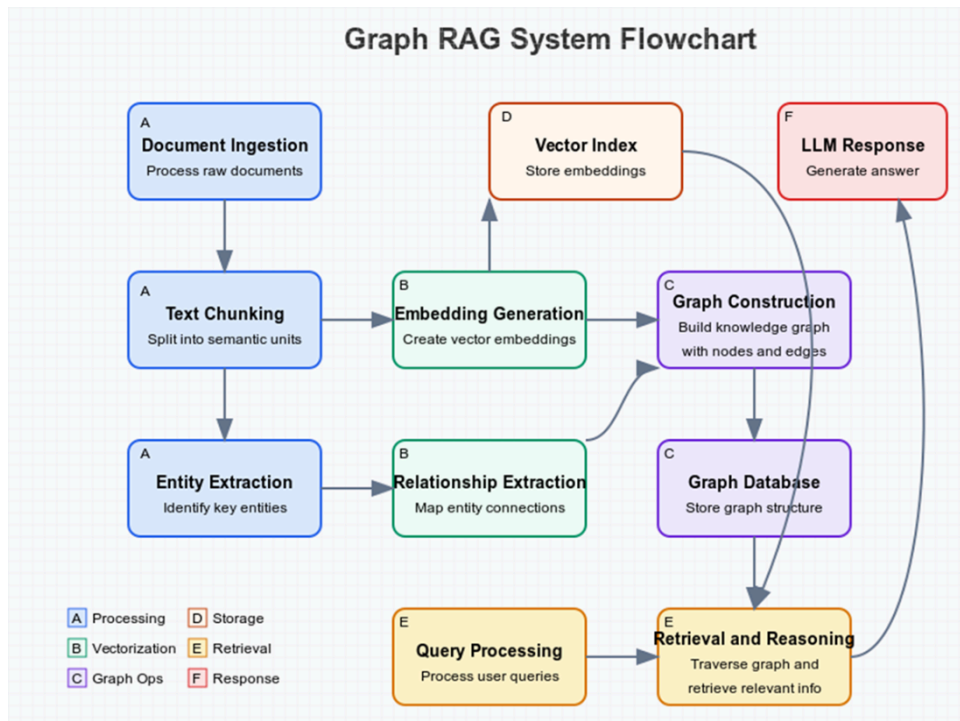


Figure 7.5 shows the workflow of a Graph RAG system.

DOCUMENT PROCESSING & KNOWLEDGE EXTRACTION

The first step is Document Ingestion, where raw documents from various sources are collected and preprocessed. These documents then undergo Text Chunking, which divides them into semantically meaningful units optimized for retrieval. After chunking, two parallel processes occur. First, Entity Extraction identifies key entities (people, organizations, concepts, etc.) within the text chunks. Second, Embedding Generation creates vector representations of each text chunk using language models, capturing their semantic meaning in a high-dimensional space.

KNOWLEDGE ORGANIZATION

The extracted information is then organized into two complementary structures: a Vector Index and a Graph Construction. The Vector Index stores the embeddings of each text chunk in a searchable database optimized for similarity searches, typically utilizing techniques like approximate nearest neighbor algorithms. Meanwhile, identified entities and their relationships flow through Relationship Extraction, where connections between entities are mapped based on contextual analysis. This data feeds into Graph Construction, which builds a knowledge graph with entities as nodes and relationships as edges. The complete graph structure is stored in a Graph Database designed for efficient traversal and relationship queries.

QUERY PROCESSING & RESPONSE GENERATION

When a user submits a query, the Query Processing component analyzes it to understand the information needs and transforms it into both a vector representation and a graph-compatible format. The Retrieval & Reasoning component then performs a hybrid search strategy:

1. It conducts similarity searches in the vector space to find relevant text chunks
2. It traverses the knowledge graph to identify related entities and relationships
3. It combines these results to create a comprehensive context that includes both directly relevant content and related information connected through the graph structure.

Finally, this enriched context is passed to the LLM Response component, which generates an answer that benefits from direct textual relevance and the structural knowledge captured in the graph relationships.

PROMPTING

Graph RAG enables richer reasoning paths. Prompts here often benefit from multi-step instructions, such as:

"Use the relationships between entities to explain how they influence each other." You can also prompt for tracing or explanations using node/edge context: "Explain how changes in oil prices affect currency value, using connected events or entities."

7.2.3 Building a Graph RAG system

Let's look at a programmatic implementation of the Graph RAG system. For simplicity and local development, we will use the ChromaDB vector database in this example. ChromaDB is lightweight, easy to set up, and well-suited for small to medium-scale RAG systems. It supports fast approximate nearest-neighbor search over text embeddings and integrates easily with Python-based workflows like the one shown here. In production-scale Graph RAG systems, Milvus is often a preferred alternative. It is designed to handle large volumes of high-dimensional vector data efficiently, making it ideal for use cases involving millions of entities or documents. Milvus also supports horizontal scaling, hybrid queries that combine vector similarity with metadata filtering, and high-throughput retrieval — all of which are valuable in complex knowledge graph scenarios. In this chapter, however, our focus is on demonstrating the Graph RAG concept using manageable data and reproducible code. Listing 7.5 shows the first step of our implementation: data preparation using the Reuters dataset.

Listing 7.5 shows step 1 of the Graph RAG implementation.

```
import os
import nltk
import numpy as np
import networkx as nx
import re
from chromadb import PersistentClient
from chromadb.config import Settings
from sentence_transformers import SentenceTransformer
from nltk.corpus import reuters
from dotenv import load_dotenv
import openai
# -----
# Setup environment
# -----
load_dotenv()
os.environ["TOKENIZERS_PARALLELISM"] = "false"
# -----
# Step 1: Load Reuters Dataset
# -----
def load_documents(limit=500):
    nltk.download('reuters')
    fileids = reuters.fileids()[:limit]
    docs = [reuters.raw(fid) for fid in fileids]
    print(f"Using {len(docs)} Reuters documents for processing.")
    return docs
```

In step 1, shown in Listing 7.5, we download the first 500 documents from the Reuters corpus provided by NLTK that form the knowledge base for the Graph RAG system. We access the list of available document IDs (fileids) and then read the raw text content of each document. Next, we need to create an embedding generation using sentence transformers and then create a graph to store the embeddings and related documents. We use NetworkX, a Python library for creating and analyzing graphs, to construct the similarity graph from our document embeddings. Each document becomes a node, and edges represent semantic similarity between documents based on their vector representations. NetworkX makes it easy to build and manipulate graphs for downstream tasks, such as clustering and traversal.

Listing 7.6 shows the implementation of embeddings and graph construction.

Listing 7.6 shows steps 2 & 3 of the Graph RAG implementation.

```
# -----
# Step 2: Generate Embeddings
# -----
def embed_documents(docs):
    print("Generating embeddings...")
    model = SentenceTransformer('all-MiniLM-L6-v2')
    embeddings = model.encode(docs, show_progress_bar=True)
    return model, embeddings

# -----
# Step 3: Build Similarity Graph
# -----
def build_similarity_graph(docs, embeddings, k=5):
    print("Constructing similarity graph...")
    G = nx.Graph()
    for i in range(len(docs)):
        G.add_node(i, text=docs[i], embedding=embeddings[i])
    for i in range(len(docs)):
        sim = np.dot(embeddings, embeddings[i])
        similar_indices = np.argsort(sim)[: -1][1:k+1]
        for j in similar_indices:
            G.add_edge(i, j, weight=sim[j])
    print(f"Graph has {G.number_of_nodes()} nodes and {G.number_of_edges()} edges.")
    return G
```

Listing 7.6 uses a pre-trained SentenceTransformer model (all-MiniLM-L6-v2) to convert each document's text into a high-dimensional embedding vector. Embeddings are numeric representations of text that capture semantic meaning, making it possible to measure similarity between documents. The embeddings for all 500 documents are computed and stored, preparing them for graph construction based on their semantic similarity. The similarity graph is created using NetworkX. Each document is added as a node in the graph with its corresponding text and embedding. Then, we find the top 5 nearest neighbors for each document by calculating cosine similarity between its embedding and all other embeddings. Edges are created between the document and its similar neighbors, with the edge weights representing their similarity scores. The result is a dense graph structure with tightly connected semantically related documents. Next, we will see how community detection is done in Listing 7.7.

Listing 7.7 shows step 4 of the Graph RAG implementation.

```
# -----
# Step 4: Community Detection
# -----
def detect_communities(G):
    print("Detecting communities...")
    from networkx.algorithms.community import greedy_modularity_communities
    communities = list(greedy_modularity_communities(G))
    print(f"Found {len(communities)} communities.")
    for idx, community in enumerate(communities[:3]):
        print(f"Community {idx+1}: {len(community)} nodes")
```

In Listing 7.7, we see that after constructing the graph, the code applies a greedy modularity-based community detection algorithm to identify clusters (or "communities") of closely related documents. These communities are groups of documents that are more interconnected with each other than with the rest of the graph. This helps organize the dataset into meaningful thematic clusters. Next, we will see how indexing is done in a Graph RAG system.

Listing 7.8 shows steps 5 & 6 of the Graph RAG implementation.

```
# -----
# Step 5: Indexing with ChromaDB
# -----
def index_with_chromadb(embeddings, docs):
    print("Setting up ChromaDB...")
    client = PersistentClient(path=".chromadb_data")
    collection = client.get_or_create_collection("graph_rag_reuters")
    ids = [str(i) for i in range(len(docs))]
    collection.add(
        ids=ids,
        documents=docs,
        embeddings=embeddings.tolist()
    )
    print("Documents indexed in ChromaDB.")
    return collection

# -----
# Step 6: Query ChromaDB
# -----
def query_chromadb(collection, model, docs, query_text):
    print(f"Running query: '{query_text}'")
    query_embedding = model.encode([query_text]).tolist()[0]
    query_results = collection.query(
```

```

    query_embeddings=[query_embedding],
    n_results=5,
    include=["documents", "distances"]
)
retrieved_info = []
for idx, doc in enumerate(query_results["documents"][0]):
    retrieved_info.append({
        "id": query_results["ids"][0][idx],
        "distance": query_results["distances"][0][idx],
        "text": doc
    })
print("ChromaDB search results:")
for info in retrieved_info:
    print(f"ID: {info['id']} - Distance: {info['distance']:.4f}")
return retrieved_info

```

Listing 7.8 shows how we perform indexing using a local ChromaDB instance. ChromaDB is a lightweight, developer-friendly vector database designed for efficient similarity search over embedded text. In this step, we create a collection, assign unique IDs to each document, and store both the documents and their corresponding embedding vectors. After indexing, we demonstrate vector search by encoding a sample query (e.g., “Financial market trends and investment news”) into an embedding using the same model as before. We then query ChromaDB to retrieve the top 3 most semantically similar documents. The result includes the content of each document, its similarity score, and its ID. This example illustrates how to retrieve relevant chunks from a corpus based on vector similarity — a core operation in any RAG system. In the next step, we use these retrieved documents as context and prompt ChatGPT to generate a final answer.

Listing 7.9 shows the last step of a Graph RAG implementation.

```

# -----
# Step 7: Generate Response with OpenAI GPT
# -----
def generate_response_with_gpt(query_text, retrieved_info):
    context_snippets = "\n\n".join([
        f"Document ID {info['id']}: \n{info['text'][:300]}..." for info in
retrieved_info
    ])
    prompt = f"""
We have retrieved a few documents based on the query: "{query_text}".
Below are summaries extracted from the top matching documents:
{context_snippets}
Please provide a clear, concise summary that combines the key insights from these

```

```

documents regarding financial market trends and investment news.
"""
    print("Generating final response using GPT-4o...")
    try:
        response = openai.chat.completions.create(
            model="gpt-4o",
            messages=[
                {"role": "system", "content": "You are an expert summarizer for
financial topics."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.7,
            max_tokens=400
        )
        return response.choices[0].message.content
    except Exception as e:
        print("OpenAI API error:", e)
        return ""

# -----
# Execution
# -----

def main():
    try:
        docs = load_documents()
        if not docs:
            print("No documents loaded. Exiting.")
            return
        model, embeddings = embed_documents(docs)
        G = build_similarity_graph(docs, embeddings)
        detect_communities(G)
        collection = index_with_chromadb(embeddings, docs)
        if not collection:
            print("Failed to index documents. Exiting.")
            return
        query = "Analyze how oil price fluctuations have influenced stock
markets, currency valuations, and international trade over the past decade.
Highlight the causal relationships among these factors. Can you also highlight
the points to support the response?"
        retrieved_info = query_chromadb(collection, model, docs, query)
        if not retrieved_info:
            print("No documents retrieved. Exiting.")
            return
        final_response = generate_response_with_gpt(query, retrieved_info)
        print("\n--- Final Answer ---\n")

```


Based on the documents provided, the summaries highlight the interconnectedness of oil price fluctuations with various financial and economic factors:

1. **Oil Prices and Economic Recovery**: Document 33 discusses how Kuwait's economic recovery, heavily reliant on oil, is significantly influenced by oil prices. This underscores the causal relationship where oil price stability or volatility can directly impact the economic trajectory of oil-dependent nations.
2. **Oil Demand and Market Dynamics**: Document 272 indicates a decrease in U.S. oil demand, contributing to market dynamics that can influence both domestic and global oil prices. Fluctuations in demand can lead to changes in oil prices, which in turn affect stock markets and trade balances.
3. **Currency Valuations**: Document 332 highlights how geopolitical and economic communications, such as the G-7 communique, can affect currency valuations, using the yen as an example. This demonstrates the influence of international political decisions on currency markets, which can be further complicated by oil price changes affecting trade balances and investor sentiment.
4. **Commodity Prices and Market Impact**: While not directly related to oil, Document 387's mention of rising lead prices due to physical market balances illustrates how commodity price fluctuations can have ripple effects in financial markets, similar to the impact of changing oil prices.
5. **Corporate Financial Performance**: Document 292 references Burmah Oil's financial performance, indicating how oil companies' profitability is closely tied to oil price trends. Higher oil prices typically lead to increased revenues and profitability for oil companies, which can drive stock market performance in the energy sector.

In summary, these documents collectively highlight that oil price fluctuations impact stock markets through changes in corporate profitability, influence currency valuations through trade balances and geopolitical factors, and affect international trade by altering demand and supply dynamics.

Listing 7.10 displays the response from Graph RAG, you may notice that the final response generated by this Graph RAG setup looks nearly identical to the output from the Naïve RAG system. This is expected — in our current implementation, each article is represented as a single node in the graph. That means we're still operating at a document level, not at the finer-grained level of entities or concepts. A fully expressive knowledge graph would extract and connect entities across articles (e.g., "oil prices", "stock market", "currency valuation"), enabling multi-hop reasoning and richer traversal paths.

For simplicity, our graph uses a one-node-per-article design to illustrate the basic mechanics of Graph RAG. In production or more advanced research settings, you'd typically build graphs from entity-relationship triples or use external ontologies to populate the graph. This difference in structure directly affects the system's ability to produce differentiated answers, which is why our example closely mirrors Naïve RAG behavior.

Let's look at the pros, cons, and use cases of the Graph RAG system.

7.2.4 Pros, Cons, and Use Cases

PROS

Graph RAG systems offer several significant advantages over traditional vector-only RAG approaches. First, they excel at preserving and leveraging relational information between entities, which vector embeddings alone typically struggle to capture. This relational context allows for a more nuanced understanding of complex topics where connections between concepts are crucial.

- **Multi-hop reasoning:** Another major strength is their ability to perform multi-hop reasoning. By traversing the graph structure, these systems can connect information across multiple documents that share related entities but might not be semantically similar enough to appear in vector-based retrieval. This enables answering questions that require synthesizing information from numerous disconnected sources.
- **Enhanced Explainability:** The explicit representation of entities and relationships simplifies tracing how information is derived and why specific content is retrieved. This transparency allows users to understand and verify the system's reasoning path, which is especially valuable in domains requiring accountability. Additionally, these systems can reduce hallucinations by grounding LLM responses in explicit knowledge structures. When information must be traversed through defined relationships in a graph, the model is more constrained to follow established knowledge patterns rather than generating plausible but incorrect connections.

CONS

Despite their advantages, Graph RAG systems face several challenges.

- **Complex Pre-processing:** They typically require more complex preprocessing pipelines than vector-only approaches. Entity extraction, relationship identification, and graph construction all add computational overhead and potential points of failure in the workflow.

- **Maintenance:** Knowledge graphs are more difficult to maintain and update than vector indices. When new information that contradicts or modifies existing relationships arrives, ensuring consistency in the graph becomes challenging. If not carefully managed, this can lead to staleness or contradictions within the knowledge base.
- **Dependency on relationships:** These systems also depend on the quality of entity and relationship extraction. If the underlying NLP components fail to identify key entities or relationships, the graph will have gaps that affect downstream reasoning. This creates a potential cascade of errors that can be difficult to diagnose and address.
- **Implementation complexity** poses another significant barrier. Building and optimizing a Graph RAG system demands expertise in both vector embedding technologies and graph databases, as well as the integration challenges of connecting these components with modern LLMs.

USE CASES

Graph RAG systems excel in areas with complex, interconnected information, where understanding relationships is crucial. Medical and healthcare applications gain significantly from this approach, as medical knowledge encompasses intricate relationships among symptoms, conditions, treatments, contraindications, and research findings that pure vector similarity might overlook.

- **Enterprise knowledge management** represents another ideal use case. Organizations with vast repositories of interrelated documents about products, projects, people, and processes can leverage Graph RAG to connect information across departmental silos and document types, providing more comprehensive answers to business questions.
- **Scientific research** and academic domains also benefit substantially. Graph RAG's ability to trace citations, connect research findings, and identify relationships between concepts across papers makes it particularly valuable for literature reviews and hypothesis generation tasks.

Financial and legal applications capitalize on Graph RAG's ability to trace complex relationships between entities, regulations, contracts, and precedents. This helps in compliance monitoring, risk assessment, and legal research, where understanding the network of relationships is often as important as the content itself.

Finally, complex customer support scenarios benefit from Graph RAG's capability to connect product information, troubleshooting steps, user history, and known issues in a cohesive knowledge structure that offers more comprehensive and contextually relevant assistance.

The ideal implementation combines the strengths of both approaches by using vector similarity for efficient initial retrieval and leveraging graph structures for relationship-aware reasoning and contextual enrichment. This creates a more powerful and flexible knowledge retrieval system than either approach could provide alone.

Graph RAG is particularly valuable in use cases where accuracy, traceability, and reasoning across linked information are critical. Unlike Naïve RAG, which retrieves isolated chunks, Graph RAG leverages structured relationships between entities, allowing the system to perform multi-hop retrieval (e.g., finding indirect causal links or dependencies). This makes it a strong fit for high-stakes domains, such as legal research, medical literature, compliance monitoring, and scientific discovery, where incorrect or fabricated outputs can have serious consequences. Additionally, because the graph structure exposes how pieces of information are connected, Graph RAG offers greater transparency and auditability traits that are essential in regulated environments or enterprise systems that require explainable AI. The following is a quick-reference summary of the pros, and cons described above.

PROS

- Preserves relational information - Captures connections between entities that vector embeddings alone struggle with
- Enables multi-hop reasoning - Can traverse graph structure to connect information across multiple documents.
- Connects disconnected sources - Links information that might not be semantically similar enough for vector retrieval.
- Enhanced explainability - Provides transparent reasoning paths showing how information is derived.
- Supports verification - Users can trace and verify the system's reasoning process
- Reduces hallucinations - Grounds LLM responses in explicit knowledge structures.
- Enforces knowledge constraints - Models follow established knowledge patterns rather than generating plausible but incorrect connections.
- Better for complex topics - Provides more nuanced understanding where connections between concepts are crucial.
- Improved accountability - Explicit representation of knowledge paths support domains requiring verification.

CONS

- Complex pre-processing - Requires more extensive pipelines for entity extraction and relationship identification
- Difficult maintenance - Knowledge graphs are harder to update and maintain than vector indices
- Consistency challenges - New contradictory information can create inconsistencies within the knowledge base

- Dependency on relationship quality - System effectiveness relies on accurate entity and relationship extraction
- Potential for cascading errors - Gaps in entity or relationship identification affect downstream reasoning
- Implementation complexity - Requires expertise in both vector embedding technologies and graph databases
- Integration challenges - Connecting graph components with modern LLMs adds technical complexity
- Higher computational overhead - Graph construction and traversal add processing requirements
- Specialized expertise needed - Development demands knowledge of both RAG and graph technologies

7.3 Hybrid RAG

Hybrid RAG systems are designed to overcome a key limitation in both Naïve and Graph RAG architectures: relying on only one retrieval signal — either dense (semantic) or sparse (keyword-based). Hybrid RAG combines the strengths of both: dense retrieval using embeddings (e.g., vector search) and sparse retrieval using traditional techniques, such as BM25. This dual-retriever setup enhances recall, robustness, and precision, particularly in scenarios where relevant documents employ different terminology from the query or when exact phrasing is crucial. In this section, we'll explore how Hybrid RAG integrates both retrieval strategies, when it makes sense to use them together, and how to build such a system using open-source tools. The term "Hybrid RAG" generally refers to systems that blend multiple retrieval strategies. There are two main forms:

- **Dense + Sparse Hybrid:** Combines semantic vector retrieval (e.g., using embeddings) with sparse keyword-based methods like BM25 or TF-IDF. This pairing improves coverage when queries and documents use different terminology or require exact matches.
- **Vector + Graph Hybrid:** Merges vector retrieval with structured querying over a knowledge graph (e.g., using Neo4j), supporting reasoning over explicit relationships.

Both approaches aim to enhance retrieval robustness and reduce blind spots. In this chapter, we primarily focus on the Dense + Sparse hybrid but also explore how graph structures can be incorporated for added structure and traceability. Before we dive deeper let's look at Figure 7.6 that shows how a Hybrid RAG system works.

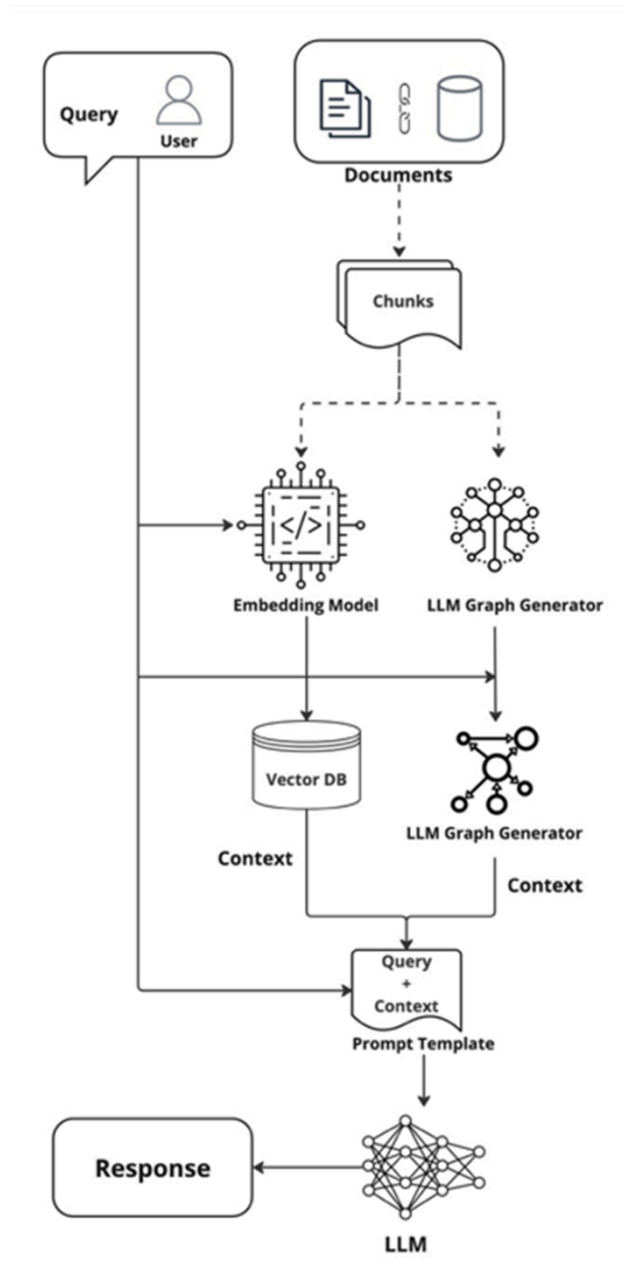


Figure 7.6 shows how a Hybrid RAG system works.

Let's dive into the components and the workflow of a Hybrid RAG system.

7.3.1 Core Components of a Hybrid RAG System

The components of a Hybrid RAG system may look similar to the Naïve RAG system. However, the difference is in how the documents/external data source is processed and how the LLM interacts with the query and the context retrieved from the documents. Let's walk through each component in depth and understand how they work and how they differ from a Naïve RAG system.

DOCUMENT COLLECTION AND KNOWLEDGE BASE

As in previous RAG systems, Hybrid RAG begins with document ingestion, chunking, and embedding. These steps create a searchable representation of the knowledge base using both vector and sparse indexing methods. The key difference in Hybrid RAG is that it maintains two parallel retrieval views of the same data: one for semantic similarity (dense vector search) and one for exact or keyword-based match (e.g., BM25). This enables broader recall and more robust retrieval in cases where a single method may miss relevant content.

RETRIEVAL SYSTEM

This component analyzes content to identify entities, concepts, and their relationships, constructing knowledge graphs that connect related information. It may employ specialized models to extract structured information from text, creating a network representation of knowledge with nodes (entities/concepts) and edges (relationships). This process transforms flat text into an interconnected web of information that preserves conceptual relationships. The system stores entity relationships and connections between pieces of information in a structure that enables traversal-based queries along relationship paths. This maintains hierarchical and associative relationships between concepts, allowing the system to follow chains of reasoning or related topics. The graph structure may implement ranking algorithms to determine the most relevant paths for a given query.

QUERY PROCESSING SYSTEM

When a user submits a query, the system transforms it into the same vector space as the document chunks, enabling similarity matching against stored vector embeddings. This typically uses the same embedding model as the document processing step to ensure consistency in the vector space, allowing for accurate similarity comparisons. Next, the context aggregation system collects and ranks results from both retrieval methods, determining which context sources to prioritize based on the query's type and characteristics. This may involve implementing weighting algorithms for different retrieval sources and resolving conflicts when various methods return contradictory information. The aggregation process aims to create a comprehensive yet relevant set of contexts for response generation.

Lastly, prompt engineering takes center stage. The prompt template component formats the user query and retrieved context into a structured prompt that guides the LLM in generating an appropriate response. The engine may apply different templates based on query classification, optimizing the context arrangement for LLM comprehension while controlling token usage and prioritizing the most relevant information.

GENERATION LAYER

The LLM, the core generation component, processes the combined query and context to produce coherent, informative responses. It leverages the retrieved information while applying its parametric knowledge, maintaining coherence between question and answer. Some implementations may include self-consistency checks or explicit reasoning steps to improve output quality. After initial generation, the system may implement post-processing to ensure factual accuracy, possibly including recursive retrieval for follow-up information needs. Some advanced systems might re-rank or filter final outputs based on confidence metrics, ensuring the delivered response meets quality thresholds.

Let's examine the workflow of a Hybrid RAG system and understand how it enables better responses via a hybrid retrieval mechanism.

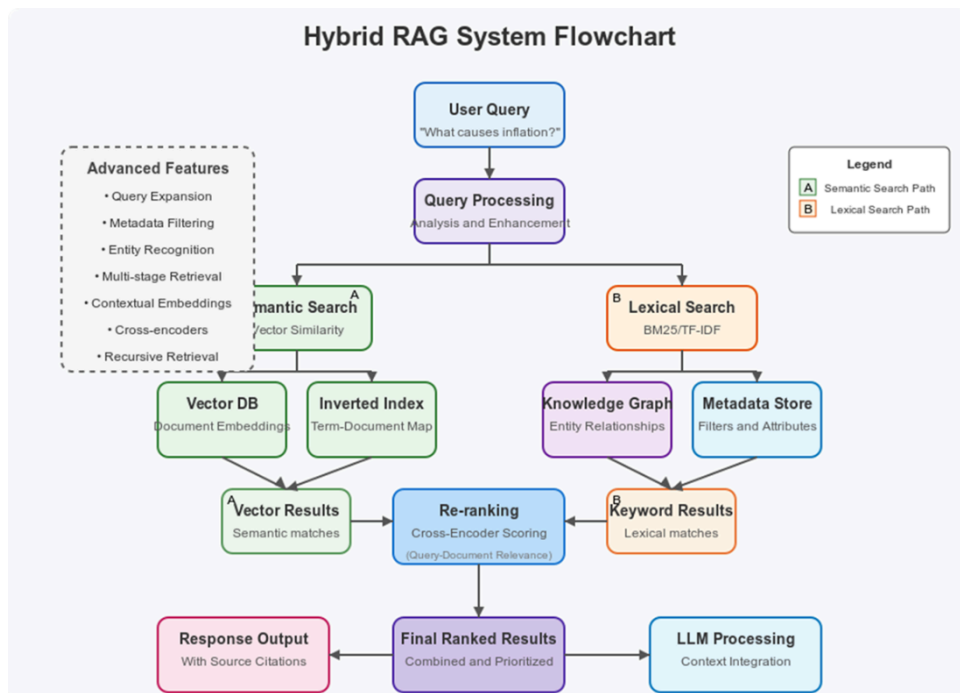


Figure 7.7 shows the workflow of a Hybrid RAG system.

7.3.2 Workflow of a Hybrid RAG System

The foundation of a hybrid RAG system remains similar to its naive counterpart: a knowledge base of documents that provides external information to the language model. However, hybrid systems build upon this foundation by implementing multiple parallel retrieval pathways and integrating advanced processing techniques at various stages of the pipeline. Let's take a closer look at the workflow illustrated in Figure 7.7.

QUERY PROCESSING & ENHANCEMENT

In a Hybrid RAG system goes beyond simple keyword matching. When a user submits a query, the system can transform it into both a dense embedding (for semantic similarity search) and a keyword-weighted form (for sparse retrieval, such as BM25). In some implementations, the query may also undergo expansion or reformulation — for instance, by adding synonyms or related concepts to improve recall across both search modes. This dual-path setup allows Hybrid RAG to retrieve documents that are either conceptually related (via embeddings) or lexically precise (via keyword overlap), maximizing the system's ability to find relevant information.

PARALLEL RETRIEVAL PATHWAYS

In Figure 7.7, we observe that parallel retrieval pathways are employed. This is the defining characteristic of a Hybrid RAG system. How does this benefit us? Using multiple retrieval paths in a hybrid RAG system significantly enhances its effectiveness by expanding its coverage and improving its robustness. Different retrieval strategies, such as dense (semantic) search, sparse (keyword) search, and structured data queries, each have their own strengths and weaknesses. By combining them, the system can capture a wider range of relevant information: both meaning-based and exact matches, ensuring that important details are not missed. This diversity also helps to make the system more resilient; if one retrieval method fails or performs poorly for a particular query, others can still deliver useful results. Additionally, multiple paths allow for better ranking and fusion of retrieved content, which means the system can prioritize the most relevant, reliable, and contextually appropriate information. Finally, this setup enables the system to handle a wide variety of question types more effectively, from precise factual queries to broader, open-ended discussions, enhancing the overall user experience by making it more accurate and adaptable. In the workflow shown in Figure 7.7, we see semantic and lexical search workflows that help make the retrieval more precise and relevant.

Semantic Search leverages vector embeddings to capture meaning beyond literal text matching. It excels at understanding conceptual relationships and can find relevant documents even when they employ different terminology than the query. This method relies on a vector database that contains document embeddings generated by neural models trained to understand semantic meaning. Lexical Search, on the other hand, focuses on keyword matching and statistical relevance. This approach uses techniques like BM25 or TF-IDF to find documents containing specific terms, technical vocabulary, or exact phrases. It operates on an inverted index that maps terms to document locations. This pathway performs well when queries involve specific technical terms, proper nouns, or exact phrases that must be matched.

Knowledge Graph Search, when available, retrieves information based on entity relationships and structured data. This approach enables the system to find information through connections between entities rather than just textual similarity. For instance, a query about "Barack Obama's education" might leverage knowledge graph connections between the person entity "Barack Obama" and education-related entities. Lastly, Metadata Filtering applies constraints based on document properties such as source, date, document type, or domain-specific attributes. This method enhances relevance by limiting results to appropriate subsets of the knowledge base. For example, a financial advice system might filter results to include only recent documents when answering questions about current tax regulations.

ADVANCED RE-RANKING

Why do we need re-ranking? A hybrid RAG system utilizes multiple retrieval pathways. When various retrieval paths are involved, such as dense retrieval, sparse retrieval, and structured searches, the system gathers a broad and diverse set of candidates. However, not all retrieved results are equally useful. Some may be loosely related, others may contain redundant or low-quality information, and some may directly and precisely address the user's intent. Re-ranking applies additional scoring, often using a more sophisticated model like a cross-encoder or LLM, to sort these results so that the most relevant and helpful content rises to the top before being passed into the generation step. Figure 7.7 shows a cross-encoder being used. However, you could leverage other methods too. Let's get to know a few of them.

- **Cross-encoder** directly assesses the relevance between query-document pairs more accurately than the embedding models used in initial retrieval. While computationally expensive (which is why they're used for re-ranking rather than initial retrieval), cross-encoders can significantly improve result quality by deeply analyzing the relationship between each document and the specific query.
- **Ensemble methods** strategically combine scores from various retrieval methods, assigning appropriate weight to semantic similarity, keyword matching, and other factors based on the characteristics of the query and the properties of the documents. These methods may dynamically adjust their weighting based on the type of query or results analysis.

- **Reciprocal Rank Fusion** or similar algorithms that intelligently merge multiple result sets while prioritizing documents that appear high in multiple lists. This approach leverages the complementary strengths of different retrieval methods, operating on the principle that documents ranked highly by multiple retrieval approaches are more likely to be relevant.

Overall, re-ranking greatly improves the quality of the final answer because it ensures that the model generates outputs based on the best available evidence rather than simply the first or most superficially similar documents. Without re-ranking, the generation model might be forced to work with noisy, incomplete, or less reliable information, which can lead to hallucinations or incorrect outputs. By carefully selecting and ordering the retrieved documents, re-ranking enhances the RAG system's accuracy, trustworthiness, and alignment with the user's intent. It effectively serves as a quality control layer between retrieval and generation, which is especially important when using hybrid retrieval methods where results come from very different sources.

CONTEXT SELECTION AND ASSEMBLY

After retrieval and re-ranking, hybrid RAG systems employ more sophisticated context selection strategies. They are:

- **Redundancy Reduction** techniques identify and eliminate duplicative information from retrieved documents, maximizing the diversity of context within the limited context window.
- **Relevance Segmentation** may further divide documents so that they focus only on the most pertinent passages rather than include entire document chunks that might contain irrelevant information.
- **Strategic Organization** arranges retrieved information to help the LLM process it effectively, for example, presenting the most relevant content first, grouping related information, or ensuring a logical flow of context.
- **Citation Tracking** maintains links between selected information and source documents, enabling the system to provide attributions in its final responses.

RESPONSE GENERATION

Hybrid RAG systems often feature more sophisticated prompt patterns and techniques that aid in generating responses. This is because Hybrid RAG systems are generally better suited for more complex problems that require breaking them into sub-problems and providing more structured responses and sources of information from which the responses were derived. An example of a prompting technique used in a RAG system is the Chain-of-Thought prompting technique. This technique encourages the LLM to reason step-by-step through the available information, improving comprehension of complex contexts. Multi-step Reasoning breaks complex queries into sub-questions that can be answered individually before being synthesized into a complete response.

Additionally, structured output formatting provides explicit instructions for how the LLM should format its response, including citations, confidence levels, or specific sections. This helps generate accurate, cited, and relevant responses for queries to the system. Lastly, you can also use feedback mechanisms to help track LLM-identified gaps in retrieved information. The feedback mechanism can also trigger additional retrieval operations to fill the gaps identified.

PROMPTING

Since Hybrid RAG combines dense and sparse signals, prompts can reinforce relevance ranking. For example:

"Answer using both key terms and general context. Prioritize facts that appear multiple times." You may also structure prompts to clarify when exact matches matter (e.g., compliance, law).

WHY RAG IMPROVES ACCURACY AND REDUCES HALLUCINATIONS

Retrieval-Augmented Generation reduces hallucinations by grounding the language model's response in external, verifiable content. Instead of relying solely on the LLM's internal parametric memory — which may be outdated, incomplete, or generalized — RAG injects real-world context into the prompt from a trusted document store. This limits the model's tendency to "fill in the blanks" with plausible sounding but incorrect information.

In Graph RAG, accuracy improves further because entity relationships and paths provide structured context that the LLM can reason over, filtering out irrelevant data and enabling multi-hop inference. In Hybrid RAG, combining sparse keyword match with dense similarity retrieval adds redundancy and complementary signals. When both retrieval paths converge on the same documents or facts, the model is more likely to focus on high-confidence, relevant evidence, reducing factual drift.

7.3.3 Building a Hybrid RAG system

Graph databases like Neo4j are ideal for representing and querying structured knowledge derived from text — such as entities and their relationships. While our Graph RAG example uses NetworkX for in-memory graphs, production systems often use Neo4j to scale up and persist graphs.

What is Neo4j? Neo4j is a graph database that stores information as nodes (entities) and edges (relationships) rather than rows and columns. It's well suited for use in RAG systems that rely on structured knowledge, such as entity graphs, where you may want to query for all relationships connected to a given concept. In our Hybrid RAG implementation, we use Neo4j to store and traverse a knowledge graph built from entity-relation-entity triples extracted from documents.

Now that we understand how a hybrid RAG system works, let's examine a programmatic example and walk through each step. Listing 7.11 shows the first step of the Hybrid RAG implementation.

Listing 7.11 shows steps 1 & 2 of the Hybrid RAG system.

```
import os
import nltk
import chromadb
import networkx as nx
import re
from chromadb import PersistentClient
from neo4j import GraphDatabase
from sentence_transformers import SentenceTransformer
from nltk.corpus import reuters
from dotenv import load_dotenv
import openai

# -----
# Setup
# -----
load_dotenv()
os.environ["TOKENIZERS_PARALLELISM"] = "false"
# -----
# Step 1: Load Reuters dataset
# -----
nltk.download("reuters")
docs = [reuters.raw(fid) for fid in reuters.fileids()[:100]]
print(f"Loaded {len(docs)} Reuters documents.")
# -----
# Step 2: Generate Embeddings
# -----
model = SentenceTransformer("all-MiniLM-L6-v2")
embeddings = model.encode(docs, show_progress_bar=True)
```

Listing 7.11 shows Steps 1 and 2 of building a Hybrid RAG system. These steps are similar to what we've seen in the Naïve and Graph RAG implementations. Now, let's move into the "hybrid" part of the pipeline. In this step, we use a language model to extract (subject, relation, object) triples from the documents. These triples form the basis of a knowledge graph, where each subject and object become a node, and the relation becomes a directed edge. We store this graph in Neo4j for efficient querying and traversal during response generation.

Listing 7.12 shows steps 3, 4 and 5 of the Hybrid RAG system.

```
# -----
# Step 3: Store Embeddings in ChromaDB                                     #A
# -----
print("Setting up ChromaDB...")
```

```

chromadb_client = PersistentClient(path=".chromadb_data")
collection = chromadb_client.get_or_create_collection(name="hybridrag_reuters")
ids = [str(i) for i in range(len(docs))]
collection.add(ids=ids, documents=docs, embeddings=embeddings.tolist())
print("Documents indexed in ChromaDB.")
# -----
# Step 4: Extract Triples Using GPT                                     #B
# -----
def extract_triples(doc_text):
    prompt = f"""
Extract entity-relation-entity triples from the following text:
{doc_text[:1000]}
Format: (Entity1, Relation, Entity2)
"""
    response = openai.chat.completions.create(
        model="gpt-4o",
        messages=[{"role": "user", "content": prompt}],
        max_tokens=300,
        temperature=0.2
    )
    text = response.choices[0].message.content.strip()
    triples = re.findall(r"\(([^\,]+), ([^\,]+), ([^\,]+)\)", text)
    return triples
# -----
# Step 5: Build Knowledge Graph + Insert to Neo4j                       #C
# -----
graph = nx.Graph()
print("Building knowledge graph...")
for doc in docs[:10]: # limit for speed
    triples = extract_triples(doc)
    for subj, rel, obj in triples:
        graph.add_edge(subj.strip(), obj.strip(), relation=rel.strip())
print(f"Knowledge Graph created with {graph.number_of_nodes()} nodes and
{graph.number_of_edges()} edges.")
driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j",
"your_neo4j_password"))
def add_to_neo4j(triples):
    with driver.session() as session:
        for subj, rel, obj in triples:
            session.run(
                """
                MERGE (a:Entity {name: $subj})
                MERGE (b:Entity {name: $obj})
                MERGE (a)-[r:RELATION {type: $rel}]->(b)
            """
            )

```



```

        """
        subj=subj.strip(), obj=obj.strip(), rel=rel.strip()
    )
print("Inserting triples into Neo4j...")
for doc in docs[:10]:
    triples = extract_triples(doc)
    add_to_neo4j(triples)
print("Knowledge Graph inserted into Neo4j.")

```

#A Store embeddings in a Chroma DB instance for documents downloaded.

#B Leverage the LLM to extract entity relationships for documents that can be used for a knowledge graph generation.

#C Build the knowledge graph using the Neo4j database.

As shown in Listing 7.12, we store the embeddings and generate knowledge graphs using ChatGPT for the documents we obtained from Reuters. Once the knowledge graphs are generated, we store them in a Neo4j database. Next, we will look at steps 6, 7, and 8 of the Hybrid RAG system.

Listing 7.13 shows steps 6, 7, and 8 of the Hybrid RAG system.

```

# -----
# Step 6: Hybrid Retrieval
# -----
def hybrid_retrieve(query_text):
    query_embedding = model.encode([query_text])[0]
    vector_results = collection.query(
        query_embeddings=[query_embedding.tolist()],
        n_results=10,
        include=["documents", "embeddings"]
    )
    docs_with_scores = []
    for doc_text, doc_emb in zip(vector_results["documents"][0],
vector_results["embeddings"][0]):
        score = float(np.dot(query_embedding, doc_emb))
        docs_with_scores.append((doc_text, score))
    reranked_docs = sorted(docs_with_scores, key=lambda x: x[1], reverse=True)
[:5]
    top_docs = [doc for doc, _ in reranked_docs]
    with driver.session() as session:
        graph_results = session.run(
            """
            MATCH (a:Entity)-[r:RELATION]->(b:Entity)
            WHERE toLower(a.name) CONTAINS toLower($query) OR toLower(b.name)

```

```

CONTAINS toLower($query)
    RETURN a.name AS entity1, r.type AS relation, b.name AS entity2
    "",
    query=query_text
)
triples = [f"{record['entity1']} -[{record['relation']}]->
{record['entity2']}" for record in graph_results]
    combined_context = "\n\n".join(top_docs + triples)
    return combined_context #A
# -----
# Step 7: Final Answer Generation with GPT
# -----
def generate_summary_with_gpt(query_text, context):
    prompt = f"""
We have retrieved the following information based on the query: "{query_text}".
Context:
{context}
Please summarize and explain the causal relationships. Present key insights
clearly.
"""
    print("Generating final response using GPT-4o...")
    try:
        response = openai.chat.completions.create(
            model="gpt-4o",
            messages=[
                {"role": "system", "content": "You are a helpful analyst."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.7,
            max_tokens=400
        )
        return response.choices[0].message.content.strip()
    except Exception as e:
        print("OpenAI API error:", e)
        return "" #B
# -----
# Step 8: Execution
# -----
query_text = "Analyze how oil price fluctuations have influenced stock markets,
currency valuations, and international trade over the past decade. Highlight the
causal relationships among these factors. Can you also highlight the points to
support the response?"
print(f"Running Hybrid RAG for query: {query_text}")
retrieved_context = hybrid_retrieve(query_text)

```


7.3.4 Pros, Cons, and Use Cases

Let's start by reviewing some use cases where a Hybrid RAG system is particularly helpful. We will then examine its pros and cons.

PROS

- **Enhanced Contextual Understanding:** Hybrid systems achieve a more comprehensive contextual understanding by combining the semantic matching capabilities of vector retrieval with the relationship awareness of graph retrieval. Vector approaches capture subtle semantic similarities between a query and relevant information, while graph approaches provide critical relationship context that explains how concepts interconnect. This combined approach offers a more complete picture than either method alone.
- **Improved Coverage for Complex Queries:** Different retrieval methods excel at different query types, and the hybrid approach leverages these complementary strengths. Graph methods handle relationship-focused queries better, excelling when users ask about connections, causes, or hierarchical relationships. Vector methods perform better with semantic similarity matching, identifying relevant information even when expressed in different terms. Together, they effectively address a wider range of query types.
- **Reduced Blind Spots:** Blind spots? Well, every type of retrieval system used has its limitations. Vector retrieval may overlook important related information that doesn't share similar language patterns but is conceptually connected. Graph retrieval might miss semantically similar but differently expressed concepts without explicit relationship links. The hybrid approach minimizes these gaps, providing more comprehensive coverage of relevant information.
- **More Nuanced Responses:** Access to relationship information allows the system to better explain causal connections and implications that may not be explicitly stated in any single document. This enhances the ability to synthesize information from multiple sources, connecting facts through logical relationships. The result is particularly valuable for comparative queries that require an understanding of similarities and differences across related concepts.
- **Better Handling of Ambiguity:** When queries contain ambiguous terms or concepts, multiple retrieval methods provide complementary context for disambiguation. Graph structures can clarify entity relationships in ambiguous queries by showing how terms relate to various concepts in different contexts. Vector similarity can identify relevant information despite ambiguous phrasing by emphasizing overall semantic intent rather than specific terms.

Let's now look at the challenges and limitations of the Hybrid RAG system.

CONS

- **System Complexity:** Integrating multiple retrieval systems significantly increases the complexity of implementation and maintenance. Organizations require expertise in various retrieval methodologies to build and maintain these hybrid systems effectively. Additionally, the higher component count introduces more potential points of failure, necessitating more sophisticated monitoring and error handling.
- **Computational Overhead:** Running multiple retrieval systems in parallel increases processing time and resource requirements, resulting in higher infrastructure costs compared to single-method approaches. The extra processing may affect response latency, creating trade-offs between response quality and speed that must be managed carefully.
- **Context Integration Challenges:** Determining how to weigh and combine different retrieval results introduces significant algorithmic challenges. The system must resolve contradictions when different methods return conflicting information and decide which sources to trust. Additionally, managing total context length limitations for LLM input requires sophisticated selection and summarization strategies when abundant relevant information is available.
- **Knowledge Base Maintenance:** When new information is added, hybrid systems require the updating of both vector embeddings and graph relationships, which increases the complexity of knowledge base maintenance. Graph relationships, in particular, require more structured updates to maintain consistency and accuracy. Organizations must develop processes to synchronize both systems, adding overhead to knowledge management workflows.
- **Evaluation Complexity:** With interconnected components affecting overall performance, isolating and enhancing specific parts of the system becomes increasingly challenging. The complex interactions between retrieval methods create difficulties in attributing performance changes to individual modifications. This necessitates sophisticated evaluation frameworks capable of assessing component- and system-level performance metrics.

USE CASES

- **Complex Research Questions:** Hybrid RAG systems excel at handling academic research questions requiring multi-faceted information drawn from diverse sources. They can assist with literature reviews spanning disparate domains, connecting findings across research papers and methodologies. Their ability to understand relationships makes them particularly valuable for interdisciplinary queries where connections between specialized fields might not be evident from semantic similarity alone.
- **Enterprise Knowledge Management:** Within organizations, these systems can navigate corporate documentation spanning various departments, connecting technical specifications with business processes and organizational policies. They enhance technical support for complex product ecosystems by relating symptoms to causes across product documentation. They also facilitate internal expertise location and synthesis, identifying connections between people, projects, and knowledge assets throughout the organization.
- **Medical Information Systems:** In healthcare contexts, hybrid RAG systems can analyze patient data while understanding treatment relationships and contraindications. They assist with medical literature review by identifying causal connections across research studies and clinical guidelines. These systems can also support drug interaction analysis and research by traversing complex relationship networks between medications, conditions, and physiological mechanisms.
- **Legal Research and Compliance:** Legal professionals benefit from these systems' ability to analyze case law while understanding the relationships between precedents across jurisdictions. They can navigate regulatory compliance amid various legal frameworks, correlating requirements with implementation guidance. Contract analysis becomes more sophisticated with the capacity to comprehend clause relationships and implications across different sections of legal documents.
- **Educational Systems:** In education, hybrid RAG systems support curriculum development by understanding the prerequisite relationships between concepts and skills. They enable personalized learning experiences informed by concept maps that track mastery and knowledge gaps. These systems excel at explaining complex topics with supporting facts drawn from various sources, connected through logical relationships that build understanding progressively.

In conclusion, hybrid RAG systems significantly advance knowledge retrieval and generation technology. Combining the strengths of vector-based semantic matching with the structural awareness of graph-based approaches, these systems deliver more comprehensive, accurate, and nuanced responses to complex queries than traditional single-method implementations.

While they introduce additional complexity and computational requirements, the benefits in response quality make them particularly valuable for domains with intricate informational needs that require factual accuracy and an understanding of relationships. As these systems evolve, we can anticipate further integration with other AI capabilities, more advanced context fusion techniques, and increasingly adaptable architectures that cater to the specific needs of various query types and knowledge domains. Organizations that invest in these technologies will be well-positioned to leverage their knowledge assets more effectively, providing users with more insightful and contextually aware AI-generated responses. The summary below distills the key pros and cons of Hybrid RAG into a quick-reference format. While Section 7.3.4 discusses these points in more depth, with explanations and use cases, the table offers a concise comparison for quick scanning. The following is a quick-reference summary of the pros, and cons described above.

PROS

- Enhanced contextual understanding - Combines semantic matching with relationship awareness.
- Comprehensive perspective - Offers a more complete picture than either vector or graph methods alone.
- Improved coverage for complex queries - Leverages complementary strengths of different retrieval methods.
- Better handling of relationship-focused queries - Excels with connections, causes, and hierarchical relationships.
- Strong semantic similarity matching - Identifies relevant information even when expressed differently
- Reduced blind spots - Minimizes gaps in retrieval coverage
- More nuanced responses - Better explains causal connections and implications
- Enhanced information synthesis - Connects facts through logical relationships
- Valuable for comparative queries - Understands similarities and differences across related concepts
- Better handling of ambiguity - Provides complementary context for disambiguation
- Clarifies entity relationships - Shows how terms relate to various concepts in different contexts
- Emphasizes semantic intent - Focuses on overall meaning rather than specific terms

CONS

- System complexity - Significantly increases implementation and maintenance complexity.
- Expertise requirements - Organizations need knowledge in various retrieval methodologies.
- Higher component count - Introduces more potential points of failure.

- Computational overhead - Increases processing time and resource requirements.
- Higher infrastructure costs - More expensive than single-method approaches.
- Potential latency issues - May affect response speed due to extra processing.
- Context integration challenges - Difficult to weigh and combine different retrieval results.
- Contradiction resolution - Must resolve conflicts when methods return contradicting information
- Context length management - Requires sophisticated selection strategies for LLM input.
- Complex knowledge base maintenance - Requires updating both vector embeddings and graph relationships.
- Synchronization overhead - Must develop processes to keep multiple systems in sync.
- Evaluation complexity - Difficult to isolate and enhance specific system components.
- Attribution challenges - Hard to determine which modifications affect performance.

7.4 Memory Augmented RAG System

A Memory Augmented Retrieval-Augmented Generation (RAG) system is an advanced architecture that extends the traditional RAG approach by integrating a persistent, dynamic memory into the retrieval and generation process. In standard RAG, when a query comes in, the system retrieves a fixed set of relevant documents from a static knowledge base (like a vector store). It feeds them into a generative model (such as a large language model, or LLM) to produce a response. In Memory Augmented RAG, the system has an additional memory module that continuously accumulates knowledge from past interactions, user feedback, or external updates. This memory can store structured or semi-structured information such as previous questions and answers, corrections, elaborations, or even newly generated knowledge. At retrieval time, the system pulls from the static corpus and this evolving memory, allowing for much more personalized, contextually rich, and up-to-date responses. Figure 7.8 shows a Memory RAG system.

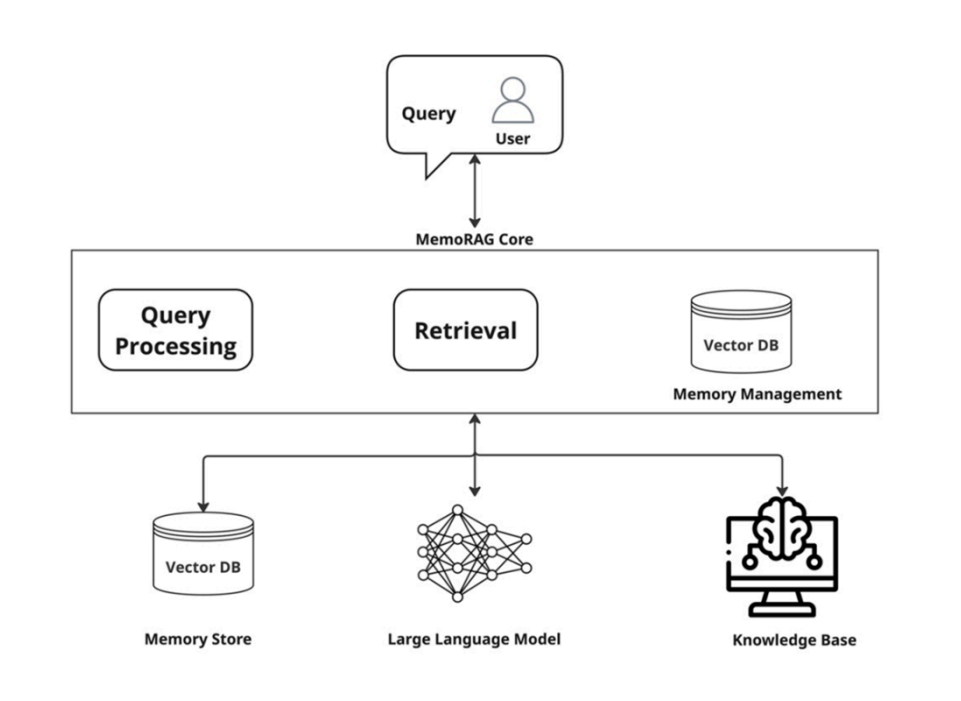


Figure 7.8 shows a Memory RAG system.

This differs from Naive RAG, which simply retrieves the top-k nearest documents using a vector similarity search without deeper structure or feedback integration. Naive RAG is effective but often brittle as it does not adapt over time or refine its retrieval strategy based on user interactions. It treats each query in isolation and depends entirely on the initial corpus quality and retriever tuning.

Graph RAG improves on this by introducing relationships between pieces of information. Instead of treating documents as independent units, it builds a knowledge graph where nodes represent entities, concepts, or documents, and edges represent relationships. Retrieval in Graph RAG is not solely based on vector similarity but also on graph traversal, enabling more semantically connected or multi-hop reasoning. For instance, if a direct answer isn't found, the model can infer it by following logical paths across the graph. Graph RAG structures information better, but still typically relies on a relatively static knowledge graph unless explicitly updated.

Hybrid RAG combines multiple retrieval methods, usually dense retrieval (vector-based) and sparse retrieval (keyword-based, like BM25) to improve recall and precision. In Hybrid RAG, the system retrieves documents using both types of methods and fuses the results to cover exact matches and semantic similarities. This makes it more robust across different query types (e.g., factual queries vs. conceptual ones) but still operates on a static or slow-changing corpus without learning from previous conversations.

In contrast, Memory Augmented RAG actively *learns* over time by adding new memory entries, adjusting memory relevance scores, or pruning outdated information — without modifying the model's internal weights. It can reflect user corrections, incorporate newly acquired facts, and build a long-term context personalized for the user or the application. In effect, it combines elements of continual learning, personalization, and retrieval, making it the most dynamic and "alive" form of RAG among the four. Let's look at the components and workflow of a Memory RAG system.

7.4.1 Components of a Memory RAG system

A Memory RAG (Retrieval-Augmented Generation) system combines traditional RAG capabilities with memory mechanisms to enable more contextual, personalized, and coherent AI interactions. The core components remain the same as those of the previous RAG systems we covered. However, a few additional components give the system a "memory." Let's look at each component.

DOCUMENT PROCESSING

Like other RAG systems, Memory RAG begins with document ingestion, parsing, and chunking. Text is cleaned, segmented into manageable units, and tagged with relevant metadata. These chunks are then embedded using a pre-trained model to enable semantic similarity search. The resulting vectors are stored in a retrieval backend, typically a vector database, such as FAISS, for fast document lookup. In contrast, the core process mirrors Naïve and Hybrid RAG systems; its outputs feed into a memory-aware pipeline where past queries and learned context can influence what gets retrieved or reinforced in future interactions.

MEMORY RAG CORE

This component forms the core of the Memory RAG system. Unlike traditional RAG systems, Memory RAG maintains multiple types of memory:

- Short-term memory for tracking the current conversation context.
- Long-term memory for persistent information about user preferences, past interactions, and domain knowledge
- Episodic memory for storing specific past interactions that can be referenced later.

This component also manages query processing, embedding generation, and vector similarity search. Depending on the implementation, we may employ techniques like hybrid search (combining semantic and keyword matching) and re-ranking to enhance retrieval quality. The final part of this component is the "Memory Manager." This component orchestrates which memories are stored, how they're organized, and when they're retrieved. It implements memory consolidation, decay, and prioritization algorithms to simulate human-like memory functions. In Figure 7.8, this is represented as "MemoRAG core".

GENERATION LAYER

This component is similar to other RAG systems that we have discussed so far. It generates responses based on the retrieved information and memories. The generation module synthesizes content, maintains coherence, and ensures that responses remain relevant to the user's query.

Next, let's look at the workflow of a Memory RAG system.

7.4.2 Workflow of a Memory RAG System

Memory RAG's approach ensures that it incorporates persistent memory mechanisms that maintain contextual awareness across multiple interactions. This approach creates more coherent, personalized, and context-aware AI responses over time. Figure 7.9 shows the workflow of a Memory RAG system.

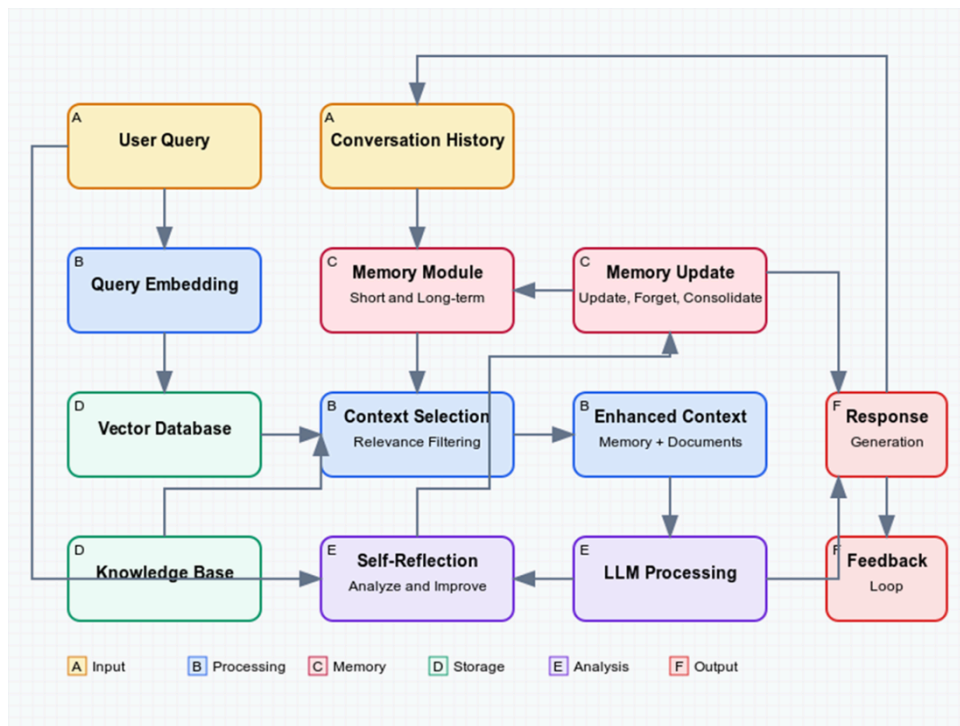


Figure 7.9 shows the workflow of a Memory RAG system.

QUERY PROCESSING

The Memory RAG workflow begins when a user submits a query. The system immediately processes this input by cleaning the text, tokenizing it, and detecting the user's intent. These preparatory steps ensure the query is standardized and structured for the subsequent stages of the pipeline. Following preprocessing, the system generates an embedding for the query. This embedding transforms the text into a numerical vector that captures the semantic meaning of the input, enabling efficient retrieval from memory.

EMBEDDING GENERATION

After preprocessing, the system generates an embedding for the user query. An embedding is a high-dimensional vector that captures the semantic content of the text in a way that can be mathematically compared to other pieces of information. The system ensures that both the incoming query and all stored memories, whether from short-term or long-term storage, reside in the same vector space. This alignment facilitates direct comparison and retrieval based on semantic similarity.

MEMORY RETRIEVAL

The system retrieves relevant information from two primary memory sources: short-term memory and long-term memory. Short-term memory contains recent conversation history, contextual clues, and transient information that helps maintain a coherent interaction. Long-term memory consists of persistent, vectorized knowledge bases that store structured information, historical data, documents, and previously learned facts. The system performs a similarity search using the query embedding to locate the most relevant memories. Retrieved results are scored and ranked to prioritize the highest quality and most relevant memories.

MEMORY FUSION AND CONTEXT ASSEMBLY

Once the system retrieves relevant memories, it fuses them into a single, enriched context. Instead of merely concatenating all retrieved information, the system intelligently filters, summarizes, deduplicates, and organizes the memories. It selects the most critical pieces of information, structures them coherently, and integrates them with the original user query. The goal is to optimize the prompt for the generation model, ensuring that the model has access to accurate, concise, and contextually appropriate knowledge when formulating its response.

RESPONSE GENERATION

The enriched context, consisting of the original query and fused memory information, is provided to the generator. The generator, typically a large language model, uses this input to craft a response that is accurate, grounded in factual retrieval, and contextually aware. The model generates a coherent answer that utilizes both the user input and the retrieved memories, producing an output that is both informative and relevant. Some systems also implement advanced generation techniques such as multi-step reasoning or tool use, but at the core, the model focuses on producing a high-quality answer from the given context.

MEMORY UPDATE AND MANAGEMENT

After generating the response, the system updates its short-term memory with the latest interaction, including the original user query and the generated response. This update ensures that future queries benefit from an increasingly rich and relevant conversational history. The system also manages short-term and long-term memories by pruning outdated information, consolidating similar entries, and prioritizing critical memories for efficient retrieval. Proper memory management is essential to maintain retrieval speed, relevance, and the quality of the system's responses over time.

7.4.3 Building a Memory Augmented RAG system.

In our Memory RAG implementation, we use LangChain's memory module to simulate conversational memory. LangChain provides built-in memory classes, such as `ConversationBufferMemory`, that enable the system to store and recall previous user queries, responses, or interaction history. This allows the LLM to respond with awareness of prior turns, maintaining coherence and continuity across multi-turn interactions. While LangChain memory isn't a complete, persistent, long-term memory system by default, it plays a key role in short-term memory management, making it well-suited for examples that demonstrate how recent context can influence the current response. Now that we understand how a Memory-Augmented RAG system works, let's examine a programmatic example and walk through each step. Listing 7.15 shows the first step of the implementation.

Listing 7.15 Step 1, and 2 of the Memory RAG implementation.

```
import os
import nltk
import numpy as np
from dotenv import load_dotenv
import openai
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS
from langchain.schema import Document
from langchain.memory import ConversationBufferMemory
# Load environment variables (for OpenAI API Key)
```

```

load_dotenv()
os.environ["TOKENIZERS_PARALLELISM"] = "false"
openai_api_key = os.getenv("OPENAI_API_KEY")
# -----
# Step 1: Load Reuters dataset                                     #A
# -----
def load_reuters_docs(limit=500):
    nltk.download('reuters')
    from nltk.corpus import reuters
    file_ids = reuters.fileids()
    docs_raw = [reuters.raw(fid) for fid in file_ids[:limit]]
    documents = [Document(page_content=doc, metadata={"source": fid}) for doc,
fid in zip(docs_raw, file_ids[:limit])]
    print(f"Loaded {len(documents)} Reuters documents.")
    return documents
# -----
# Step 2: Create vector store with FAISS                           #B
# -----
def create_vector_store(documents):
    embedding_model = OpenAIEmbeddings(model="text-embedding-ada-002")
    vector_store = FAISS.from_documents(documents, embedding_model)
    return vector_store, embedding_model

```

#A Load the Reuters dataset containing 500 financial documents

#B Create the vector database with FAISS.

Listing 7.15 shows the initial two steps of the Memory RAG implementation. These steps are similar to those of all the other RAG systems we have reviewed. Step 1 downloads and accesses the Reuters corpus from NLTK. It extracts raw text from up to 500 Reuters news articles (configurable via the limit parameter) and converts each into a LangChain Document object. Each document contains full text content and metadata, including the source file ID for traceability. This establishes the knowledge base that will be queried later.

Step 2 establishes the semantic search capability. It utilizes OpenAI's text-embedding-ada-002 model to convert each document into a high-dimensional vector representation that captures semantic meaning. These embeddings are then stored in a FAISS vector database, enabling efficient similarity search. FAISS (Facebook AI Similarity Search) is designed for quick nearest-neighbor searches in high-dimensional spaces, making it ideal for retrieving relevant documents. Next, let's look at building the core of the Memory RAG system.

In this example, we use a system prompt to define the LLM's role as a "financial assistant that provides concise, insightful answers." This kind of role-based instruction helps constrain the model's tone, style, and focus, especially important in Memory RAG, where the model responds based on both current and historical context. Prompting the model with a persona or intent improves coherence across turns and aligns the response with the user's expectations, particularly in multi-turn or domain-specific scenarios.

Listing 7.16 shows the Memory building step of the Memory RAG system.

```
# -----
# Step 3: Build the Memory RAG system
# -----
def run_queries(embedding_model, documents, queries):
    long_term_memory = []
    for query in queries:                                     #A
        # Enhance query with long-term memory context
        enhanced_query = query
        if long_term_memory:
            query_embedding = embedding_model.embed_query(query)
            memory_texts = [f"User: {m['query']}\nAssistant: {m['response']}" for
m in long_term_memory]
            memory_embeddings = [embedding_model.embed_query(text) for text in
memory_texts]
            similarities = [np.dot(query_embedding, mem_emb) for mem_emb in
memory_embeddings]
            memory_context = ""
            for idx in np.argsort(similarities)[-2:]:
                if similarities[idx] > 0.75:
                    memory_context += f"Previously discussed:\nQ:
{long_term_memory[idx]['query']}\nA: {long_term_memory[idx]['response']}\n\n"
#B
            if memory_context:
                enhanced_query = memory_context + f"\nCurrent question: {query}"
        # Embed and retrieve top-k similar documents
        query_embedding = embedding_model.embed_query(enhanced_query)
        results = FAISS.from_documents(documents, embedding_model)
        retriever = results.as_retriever(search_kwargs={"k": 3})
        retrieved_docs = retriever.get_relevant_documents(enhanced_query)
        sources = [doc.metadata['source'] for doc in retrieved_docs]
        context = "\n\n".join([f"[{doc.metadata['source']}]:
{doc.page_content[:300]}..." for doc in retrieved_docs])
        # Call OpenAI API with context + question                                     #C
        prompt = f"""
You are a financial analyst. Given the query and relevant document
```

snippets below, write a clear and concise answer.

```

Query:
{enhanced_query}
Context:
{context}
Answer:
"""
try:
    response = openai.chat.completions.create(
        model="gpt-4o",
        messages=[
            {"role": "system", "content": "You are a financial assistant
that provides concise, insightful answers."},
            {"role": "user", "content": prompt}
        ],
        temperature=0.7,
        max_tokens=300
    )
    answer = response.choices[0].message.content
except Exception as e:
    answer = f"OpenAI API Error: {e}"
# Output result
print(f"\nUser: {query}")
print(f"AI: {answer}")
print(f"Sources: {sources}")
# Store in long-term memory if long enough #D
if len(answer) > 300:
    if long_term_memory:
        similarities = [
            np.dot(query_embedding, m["embedding"]) for m in
long_term_memory
        ]
        if max(similarities) < 0.85:
            long_term_memory.append({
                "query": query,
                "response": answer,
                "embedding": query_embedding
            })
        else:
            long_term_memory.append({
                "query": query,
                "response": answer,
                "embedding": query_embedding
            })
    })

```



```

if len(long_term_memory) > 50:
    long_term_memory = long_term_memory[-50:]

```

#A Enhance the query with the long-term memory if it exists.

#B Retrieve the similarities if the long-term memory contains relevant data. Check for the similarity score and then add the data to the context.

#C Call the LLM with the query and the context.

#D Retrieve the response and then store it in the memory for future reference if it is long enough.

Listing 7.16 illustrates how we create the memory component of the Memory RAG system. In this step, we first check whether long-term memory contains interactions. If it does, the current query's response is determined based on past interactions. If a substantial similarity is detected, the system retrieves the past interactions and prepends them to the current query as context. Next, using the enhanced query, the system searches the vector database to find the three most semantically similar documents. These documents provide the factual context needed to answer the question. During the response generation phase, the system constructs a prompt that includes the query and retrieved document snippets. This prompt is sent to OpenAI's GPT-4o model, which generates a concise, informed response based on the provided context. In the last step of the Memory RAG system creation, we check if the generated answer is substantial (>300 tokens). The system evaluates whether to add it to long-term memory, storing only sufficiently dissimilar responses (similarity score <0.85) from existing memories to avoid redundancy. The memory is capped at 50 entries, with older entries discarded when this limit is reached. The way we have implemented the Memory RAG system can be changed to store more than 50 entries, check for more than just tokens, and also similarity scores.

Next, let's see how the Memory RAG system builds on its core component to respond to queries.

Listing 7.17 shows the final step of the Memory RAG implementation.

```

# -----
# Execution of MemoRAG
# -----
def main():
    documents = load_reuters_docs(limit=500)
    vector_store, embedding_model = create_vector_store(documents)
    queries = [
        "What is driving oil prices in the last decade?",
        "Analyze how oil price fluctuations have influenced stock markets,
        currency valuations, and international trade over the past decade. Highlight the
        causal relationships among these factors. Can you also highlight the points to
        support the response?"
    ]
    run_queries(embedding_model, documents, queries)
if __name__ == "__main__":

```

```
main()
```

Output:

User: What is driving oil prices in the last decade?

AI: Over the last decade, oil prices have been influenced by several key factors:

1. **Global Demand Fluctuations**: Changes in oil demand, such as the recent 2.6% decrease in U.S. oil demand, impact prices significantly. When demand falls, prices tend to decrease, and vice versa.

2. **OPEC's Role**: The Organization of Petroleum Exporting Countries (OPEC) plays a crucial role in stabilizing oil prices. Their ability to set production targets and manage supply levels influences global prices, as seen in their efforts to maintain stable prices for economic recovery in oil-dependent countries like Kuwait.

3. **Currency Valuations**: The value of the U.S. dollar affects oil prices, given that oil is traded globally in dollars. A weaker dollar can make oil cheaper for foreign buyers, boosting demand and prices.

4. **Supply-side Dynamics**: The availability of cheap oil feedstocks and high utilization rates in industries such as the U.S. petrochemical sector can drive demand for oil, contributing to higher prices.

Overall, the interplay between supply-demand dynamics, OPEC's strategic output decisions, currency fluctuations, and industry-specific demand are primary drivers of oil prices in the last decade.

Sources: ['test/15351', 'test/14891', 'test/15063']

User: Analyze how oil price fluctuations have influenced stock markets, currency valuations, and international trade over the past decade. Highlight the causal relationships among these factors. Can you also highlight the points to support the response?

AI: Oil price fluctuations over the past decade have significantly influenced stock markets, currency valuations, and international trade, with several causal relationships among these factors:

1. **Stock Markets**: Oil prices impact stock markets primarily through their influence on energy-related stocks and broader economic conditions. High oil prices can lead to increased costs for businesses, squeezing profit margins and potentially depressing stock prices. Conversely, lower oil prices can boost consumer spending by reducing transportation and production costs, which may positively affect stock markets.

2. **Currency Valuations**: Fluctuations in oil prices can affect currency valuations, particularly for oil-exporting and oil-importing countries. For oil-exporting nations, higher oil prices generally strengthen their currencies due to increased revenue from oil exports. Conversely, oil-importing countries may experience currency depreciation as higher oil prices increase their trade deficits.

3. **International Trade**: Oil prices directly impact the cost of transportation and production, influencing international trade balances. Higher oil prices can increase the cost of goods, potentially reducing trade volumes, while lower prices may enhance trade by making goods cheaper to produce and transport.

Overall, the interplay between oil prices and these economic factors is complex. For instance, as seen in Kuwait's economy, oil prices are crucial for economic recovery, highlighting the dependency of oil-exporting countries on stable prices. Similarly, changes in U.S. oil demand, as noted in the 2.6% decrease, can influence domestic economic conditions, further affecting international trade dynamics and currency stability.

Sources: ['test/14891', 'test/15351', 'test/15520']

In Listing 7.17, we mimic successive similar queries to the system. The Memory RAG system uses the earlier query as context for the later query and generates a response. The response is detailed in both cases. However, in the second instance, it is more detailed and contains more data to help understand it. Do you want to try and build this system and use different queries to see how it all works?

Next, let's look at the pros, cons, and use cases of this system.

7.4.4 Pros, Cons, and Use Cases

PROS

- **Persistent Context Across Sessions**: Memory augmentation enables the system to retain and recall information from past interactions. This continuity allows it to "remember" user goals, previous queries, or decisions across multiple sessions. For example, suppose a user is working on a long-term project or research task. In that case, the system can bring back prior context without repeated re-explanation, making the interaction feel more natural and human-like.

- **Improved Personalization:** With memory, the system can learn user preferences, interests, domain knowledge, and interaction patterns over time. It can fine-tune its responses based on this data, such as adjusting the level of detail in explanations, avoiding redundant suggestions, or even mirroring a user's preferred terminology or style. This capability is especially valuable in education, customer support, or productivity tools, where personalization improves engagement and effectiveness.
- **Reduced Redundant Retrieval:** Traditional RAG systems retrieve relevant documents for every query. Memory-augmented systems can store and reference previously retrieved documents or results, reducing the need for repeated calls to the retrieval backend. This saves computational resources and latency, especially when users revisit the same topics or documents multiple times.
- **Enhanced Multi-turn Coherence:** Conversations and tasks often unfold over several turns. A memory-augmented RAG can maintain coherence across a sequence of queries by keeping track of prior discussion points, entities, and references. For instance, in technical support, remembering previously discussed error messages or configuration steps helps the system avoid asking the same questions again and enables better follow-up suggestions.
- **Support for Iterative Refinement:** Users frequently revise or refine their queries as they clarify their intent. A memory-augmented system can treat each interaction as part of a growing knowledge thread. This allows for more effective iterative reasoning, accumulating details over time and producing richer, more accurate responses than starting fresh with each prompt.

CONS

- **Memory Management Complexity:** Determining what to store in memory, how long to keep it, and how to represent it meaningfully is a complex challenge. Not all information is equally valuable; irrelevant or outdated data can clutter memory. Strategies like memory pruning, summarization, or relevance scoring are needed, but tuning them correctly is non-trivial and often domain-specific.
- **Privacy and Security Concerns:** Storing user history, mainly if it includes sensitive personal, medical, financial, or corporate data, raises significant privacy risks. Systems must implement robust access controls, data anonymization, encryption, and compliance mechanisms (e.g., GDPR or HIPAA) to ensure that memory storage does not violate privacy expectations or legal standards.

- **Potential for Hallucination or Error Propagation:** If the system stores incorrect information, whether due to misinterpretation, outdated data, or hallucinations from the model, it may repeatedly reference or build upon those errors in future interactions. This can degrade user trust and introduce compounding inaccuracies, especially in high-stakes contexts like healthcare or legal advice.
- **Scalability Challenges:** As memory grows with each user and session, it can become increasingly costly to store, retrieve, and process. Scaling across thousands or millions of users demands infrastructure that efficiently segments, manages, and accesses individualized memory without latency or cost spikes. This may require sharded memory stores, caching, and distributed memory architectures.
- **Increased System Complexity:** Adding memory to a RAG architecture requires synchronization between the memory module, the retriever, and the generator. This adds operational complexity and increases the likelihood of bugs, misalignment, or output inconsistencies. Building and maintaining these systems requires NLP modeling and memory systems architecture expertise.

USE CASES

- **Long-form Creative Writing or Programming Assistance:** Writers and coders often return to the same project over multiple sessions. A memory-augmented assistant can track story arcs, character backstories, and programming constraints across interactions. For instance, a writer revisiting a novel draft can rely on the assistant to recall narrative choices, while a developer can receive help maintaining consistency in variable naming, API usage, and logic across files.
- **Customer Support Chatbots:** Customer service systems leverage memory by recalling user-specific issues, preferences, and device history. When a customer returns for assistance, the chatbot can retrieve information about their previous problems or open tickets, thereby avoiding repetitive troubleshooting and speeding up resolution. This enhances customer satisfaction and lessens the workload on agents.
- **Personalized Learning Platforms:** In educational settings, memory can track a student's progress, strengths, and weaknesses. The system can then adapt instruction to bridge knowledge gaps, reinforce previous lessons, and build upon mastered skills. This results in more effective, adaptive learning experiences and fosters long-term retention through spaced repetition and scaffolded instruction.

- **Enterprise Assistants:** Memory-augmented systems can serve as organizational knowledge aides, tracking project status, decisions made in previous meetings, or documents accessed by users. They can summarize past activities, suggest next steps, or identify team dependencies. This is invaluable in collaborative work environments where institutional memory often gets lost.
- **Therapeutic or Coaching Applications:** Digital therapy or coaching tools benefit from long-term memory that tracks emotional trends, prior sessions, and user goals. By remembering what has been discussed and the strategies suggested, these systems can provide more empathetic, consistent, and context-aware guidance, fostering a deeper sense of engagement and care.

Memory-Augmented RAG systems represent a significant advancement in the evolution of AI-driven knowledge interaction. By integrating persistent memory with retrieval-augmented generation, these systems provide essential continuity, personalization, and coherence across sessions for long-term, multi-turn, or user-specific tasks. They enable AI models to retain past context, adjust responses over time, and deliver richer, more efficient interactions that reflect user intent and history.

While adding memory presents challenges related to system complexity, privacy, and scalability, the benefits it brings to user experience and task performance are significant, particularly in fields such as education, support, creative work, and enterprise productivity. As the field progresses, we can anticipate advancements in memory management strategies, privacy-aware architectures, and intelligent memory pruning that will further improve system effectiveness. Organizations that adopt memory-augmented systems will achieve a competitive advantage by providing more intelligent, contextual, and responsive AI capabilities that adapt to users over time. The following is a quick-reference summary of the pros, and cons described above.

PROS

- Persistent context across sessions - Retains information from past interactions
- Session continuity - "Remembers" user goals and previous queries
- Reduced re-explanation - Users don't need to repeat context
- Improved personalization - Learns user preferences and interests over time
- Adaptive interactions - Adjusts detail level based on user knowledge
- Terminology adaptation - Can mirror user's preferred terminology or style
- Reduced redundant retrieval - Stores previously retrieved documents
- Resource efficiency - Saves computational resources and reduces latency
- Enhanced multi-turn coherence - Maintains discussion flow across queries
- Contextual awareness - Keeps track of prior discussion points and references

- Support for iterative refinement - Treats interactions as growing knowledge threads
- Progressive reasoning - Accumulates details over time for richer responses

CONS

- Memory management complexity - Difficult to determine what to store and for how long.
- Irrelevant data challenges - Outdated information can clutter memory.
- Need for sophisticated strategies - Requires memory pruning, summarization, and relevance scoring.
- Privacy and security concerns - Risks when storing sensitive personal data
- Compliance requirements - Must implement robust controls for GDPR, HIPAA, etc.
- Potential for hallucination - May store and propagate incorrect information.
- Error propagation - Incorrect data can lead to compounding inaccuracies.
- Trust degradation - Errors can undermine user confidence over time
- Scalability challenges - Increasingly costly to store and process growing memory
- Infrastructure demands - Requires efficient segmentation and memory access systems
- Increased system complexity - Requires synchronization between multiple components
- Technical expertise requirements - Demands both NLP and memory architecture knowledge

7.5 Which one to use?

Several factors can help decide which RAG system to use and for what use cases. Below is a table comparing all the RAG systems we went through in this chapter along with what they are best used for.

Feature/System	Naive RAG	Graph RAG	Hybrid RAG	Memory-Augmented RAG
Retrieval Type	Vector-based only	Graph-based only	Vector + Graph	Vector + Episodic Memory
Storage Layer	Vector DB	Knowledge Graph	Vector DB + Knowledge Graph	Vector DB + Long-Term Memory Store
Context Granularity	Semantic chunks	Entities, relationships, paths	Combined: semantic+ structural	Semantic+ conversational memory
Reasoning Ability	Limited to retrieved chunk content	High (multi-hop, logical, causal)	High (broad + deep reasoning)	Medium (reasoning across dialogue history)
Explainability	Low	High	Medium-High	Medium
Recall Coverage	High (semantic match)	Medium-Low (explicit links only)	High (semantic+ structured recall)	Medium-High
Personalization	None	Low	Low	High (remembers user context)
Latency	Low	Medium-High	High	Medium
Infrastructure Complexity	Low	Medium-High	High	Medium
Best For	FAQ bots, simple document Q&A	Scientific QA, logic-heavy tasks	Enterprise assistants, technical research	Conversational agents, long-session tasks

7.6 Summary

- Retrieval-augmented generation systems have evolved from simple naive implementations to sophisticated memory-enhanced architectures. Each advancement addresses specific limitations of previous approaches while maintaining the core principle of combining external knowledge retrieval with generative capabilities.
- Traditional RAG systems consist of three core components: a document indexer that processes and stores information, a retriever that fetches relevant documents based on queries, and a generator that synthesizes responses by combining retrieved context with the query, offering improved factuality but often struggling with coherence across multiple turns.

- Graph RAG enhances retrieval by representing knowledge as interconnected nodes and edges, capturing relationships between concepts rather than treating documents as isolated entities. This enables more contextually relevant retrieval through traversal of semantic connections and hierarchical information structures.
- Hybrid approaches combine various retrieval methods (dense, sparse, graph-based) and knowledge sources (vector stores, structured databases, APIs) to leverage the strengths of each paradigm, dynamically selecting the most suitable retrieval strategy based on query characteristics and available knowledge.
- Memory RAG enhances traditional models by incorporating short-term conversation history and long-term persistent memory. This enables the system to maintain context through conversation turns, remember user preferences, and build upon previous interactions without redundant retrievals.
- While naive RAG follows a linear process (query → retrieve → generate), advanced systems employ feedback loops, query reformulation, and multi-hop reasoning that iteratively refine both the retrieval and generation steps based on intermediate results.
- Naive RAG excels in straightforward question-answering scenarios, Graph RAG excels in complex reasoning tasks that require understanding of relationships, Hybrid RAG shines in domain-specific applications needing diverse knowledge sources, and Memory RAG is effective in maintaining coherent, personalized conversational experiences.
- More sophisticated RAG systems provide enhanced response quality and contextual understanding, but they come with increased computational complexity, latency, and implementation difficulties. Practitioners must balance these factors against application requirements.
- Advanced RAG systems encounter challenges such as knowledge graph construction and maintenance, effective memory management policies, retrieval strategy orchestration, and the necessity for specialized evaluation metrics that extend beyond traditional accuracy measures.

Appendix A

A.1 Appendix: Setting up LangChain

In this appendix you'll learn to set up LangChain for your machine. We cover the approaches both for Mac and for Windows.

A.1.1 Setting Up LangChain on Mac

First, ensure you have the Xcode command-line tools installed on your Mac. The developer tools help you run command-line operations in the Terminal on a Mac. To ensure that you have the latest developer tools installed run the following command.

```
xcode-select -install
```

This command will show a pop-up asking you to install the developer tools. We also need a package manager to ensure you can easily install, update, and manage various libraries that we will need to start programming with LangChain. The most common and widely used package manager is Homebrew. It is a package manager for MacOS and Linux, enabling users to manage (install, update, uninstall) various libraries, applications, and tools. We will use HomeBrew throughout this book. You can find more details on installation, etc. here: <https://brew.sh/>

Python is the most commonly and widely used programming language in AI and ML. This book will use Python for all our examples and experiments. Before that, we will install the latest version of Python using the following command.

```
brew install python
```

This command will install Python on your Mac. We can now install LangChain. Before we install LangChain, we will create a Python virtual environment. A Python virtual environment helps create efficient and organized development processes by isolating dependencies and ensuring code reproducibility. The following command will create a Python virtual environment.

```
python3 -m venv promptengg-venv
```

In the listing, we have named our virtual environment as promptengg-venv. You can choose to name your virtual environment as desired. We will now activate the virtual environment using the following command.

```
source ./promptengg-venv/bin/activate
```

This command will activate the virtual environment. Once the virtual environment is activated you can run the following commands to install LangChain and all its dependencies.

```
pip3 install LangChain
```

```
pip3 install --upgrade --quiet langchain-core langchain-community langchain-openai
```

```
pip3 install langchain_experimental
```

These commands will install all required dependencies and libraries of LangChain, which we will use in our code samples in this Chapter.

As a final step to get started with programming in this Chapter let us activate the virtual environment that we created earlier in the Chapter using the command:

```
source ./promptengg-venv/bin/activate
```

A.1.2 Setting up LangChain in Windows

Download the latest version of the Python installer from the following URL.

<https://www.python.org/downloads/>

Double-click on the Python installer and choose the option to install the launcher for all users by checking the corresponding checkbox so that all users of the computer can access Python. Also, click on the checkbox "Add python.exe to PATH". Verify the Python installation in windows by running following command.

```
Python -version
```

We can now install LangChain. Before we install LangChain, we will create a Python virtual environment. A Python virtual environment helps create efficient and organized development processes by isolating dependencies and ensuring code reproducibility. The following commands will install and create a Python virtual environment.

```
pip install virtualenv
```

```
python -m virtualenv promptengg-venv
```

In the listing, we have named our virtual environment as promptengg-venv. You can choose to name your virtual environment as desired. We will now activate the virtual environment using the following command.

```
.\promptengg-venv\Scripts\activate
```

Tip: If you encounter an UnauthorizedAccess access issue, run the following command to set execution policy as unrestricted.

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Unrestricted
```

This command will activate the virtual environment. Once the virtual environment is activated you can run the following commands to install LangChain and all its dependencies.

```
pip install LangChain
```

```
pip install langchain-community
```

```
pip install langchain_experimental
```

These commands will install all required dependencies and libraries of LangChain, which we will use in our code samples in this Chapter.