9 Prompt engineering

This chapter covers:

- Basic and advanced prompt types for different programming scenarios
- Crafting effective prompts using context, clear instructions, and examples
- Iterative approaches such as chain of thought and recursive prompting
- Context manipulation and instruction refinement for code generation
- Specialized techniques to control output format for technical documentation

Prompt engineering has become a key skill for developers using generative AI tools of any type. It facilitates communication with these powerful assistants. Instead of getting generic responses, strong prompts can greatly enhance the quality, accuracy, and usefulness of AI-generated code and documents. This chapter examines some practical techniques that will help you change vague requests into clear instructions and obtain useful responses. This way, you can maximize AI's capabilities, while keeping control over the output. You'll turn these tools into dependable partners in your development process.

Some parts of prompt engineering may seem natural—like how we learned to improve our search engine questions over time. However, learning the particulars of this skill can make a big difference in the results you get from chat models and large language model (LLM)-based programming tools.

Let's explore how to create better prompts to get the best results, save time, and increase productivity.

9.1 Understanding prompt engineering

Imagine you're trying to debug a Python function that's throwing an unexpected error. You could use a simple prompt and ask AI to

JM Fix this code.

You'd probably get back a vague or unhelpful response. Or, you could craft a more effective prompt, such as

JM I'm debugging a Python function that calculates user discounts. When processing negative values, it throws this error: `ValueError: Discount cannot be negative`.

Here's the code:

```
def apply_discount(price, discount):
    final_price = price * (1 - discount)
    return final_price
```

Please:

1. Identify the bug.
2. Explain why it occurs.
3. Provide a fixed version with input validation.
4. Add example test cases.

Here's what ChatGPT returned when I sent this prompt:

1. Identifying the Bug:

   The code does not check if the `discount` value is negative. When a negative discount is provided, the function proceeds to calculate the `final_price` without validation, which may lead to unexpected behavior or logical errors.

2. Why It Occurs:

   A negative discount effectively increases the price, which is illogical in most discount-related contexts. The error message (`ValueError: Discount cannot be negative`) mentioned in your question suggests that this case should be handled explicitly, but the function lacks such validation.

3. Fixed Version with Input Validation:

   Here's an improved version of the function with proper input validation:

````
```python
def apply_discount(price, discount):
    """
    Applies a discount to the price.

    Args:
        price (float): The original price of the item.
        discount (float): The discount as a fraction (e.g., 0.2 for 20%).

    Returns:
        float: The final price after applying the discount.

    Raises:
        ValueError: If the price or discount is invalid
(negative, greater than 1 for discounts).
    """
    if price < 0:
        raise ValueError("Price cannot be negative.")
    if discount < 0 or discount > 1:
        raise ValueError("Discount must be between 0 and 1.")

    final_price = price * (1 - discount)
    return final_price
```
````

4. Example Test Cases:

   Here are some test cases to ensure the function behaves correctly:

````
```python
def test_apply_discount():
    try:
        # Test case 1: Valid inputs
        assert apply_discount(100, 0.2) == 80.0
        print("Test case 1 passed")

        # Test case 2: Zero discount
        assert apply_discount(100, 0.0) == 100.0
        print("Test case 2 passed")

        # Test case 3: Full discount (free item)
        assert apply_discount(100, 1.0) == 0.0
        print("Test case 3 passed")

        # Test case 4: Negative price
        try:
            apply_discount(-100, 0.2)
        except ValueError as e:
            assert str(e) == "Price cannot be negative."
            print("Test case 4 passed")

        # Test case 5: Negative discount
        try:
````
````

````
            apply_discount(100, -0.1)
        except ValueError as e:
            assert str(e) == "Discount must be between 0 and 1."
            print("Test case 5 passed")

        # Test case 6: Discount greater than 1
        try:
            apply_discount(100, 1.2)
        except ValueError as e:
            assert str(e) == "Discount must be between 0 and 1."
            print("Test case 6 passed")

    except AssertionError:
        print("A test case failed.")

# Run tests
test_apply_discount()
```
````

This implementation and test cases ensure that invalid inputs are handled properly and the function produces accurate results for valid inputs.

This second prompt gives a much more useful response. Why? Because it uses the key elements of effective prompt engineering, as shown in figure 9.1.

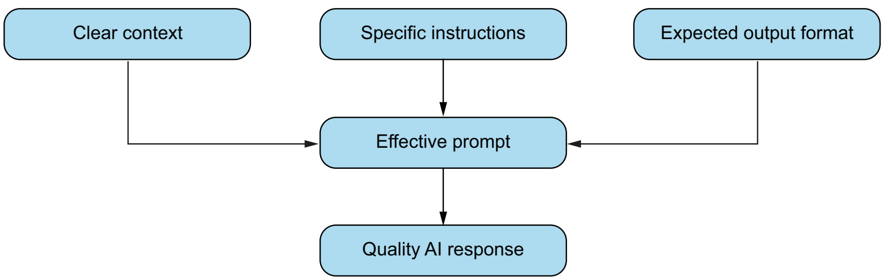

Figure 9.1 A quality prompt starts with clear context, specific instructions, and an expected format. The better the input quality, the better the result.

Prompt engineering is the skill of crafting clear, effective instructions that help AI models understand and respond to your needs accurately. Think of it as of being a good teacher—the better you explain what you want, the better results you'll get (figure 9.2).

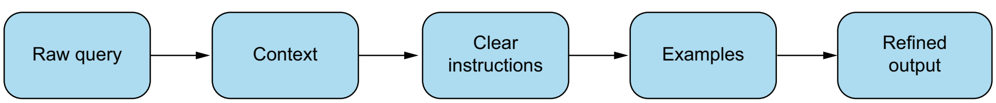

Figure 9.2 A good prompt starts as a query, and context is added. If you supply clear instructions and some examples, you can change the output drastically.
````

The following list shows some key components of an effective prompt:

- *Context*—Background information, relevant constraints, and technical requirements
- *Clear instructions*—Step-by-step requests, specific requirements, and expected format
- *Examples (when helpful)*—Sample inputs/outputs, preferred formatting, and edge cases to consider

Let's see these in action with another development scenario. Here is an example of an ineffective prompt:

JM Write a function to validate email addresses.

And here is a well-engineered prompt:

JM Create a Python function that validates email addresses with these requirements:

1. Accepts a single string parameter
2. Checks for:
   - Proper @ symbol placement
   - Valid domain format
   - Allowed characters
3. Returns boolean (True if valid)
4. Includes error handling

Please provide:

- Function implementation
- Docstring with examples
- At least 3 test cases
- Common edge cases handled

The difference is clear—the second prompt provides context, specific requirements, and expected deliverables. This structured approach consistently produces better results when working with AI coding assistants.

9.1.1 Why prompt engineering matters

Good prompt engineering

- Reduces back-and-forth with AI tools
- Generates more accurate code
- Saves debugging time
- Produces better documented solutions
- Handles edge cases more effectively

Think of prompt engineering as writing good requirements—the more specific and clear you are, the better the output. As you progress through the discussion, you'll learn increasingly sophisticated techniques to make AI tools more effective partners in your development workflow.

9.2 Understanding the anatomy of a prompt

Good prompt engineering includes good communication. Key aspects of prompt engineering incorporate providing context, clear instructions, and good examples (figure 9.3).

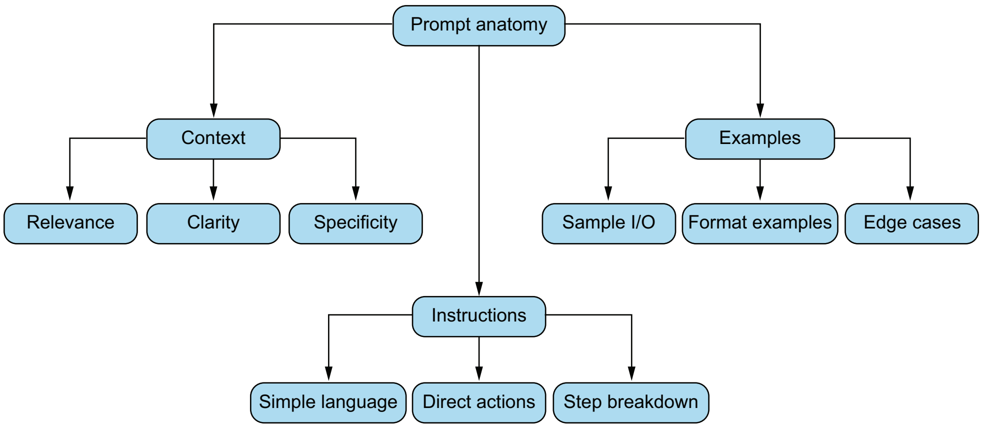

Figure 9.3 A good prompt starts with great context (usually provided by the tool). This includes your existing project. Next, you add good examples and clear instructions. The process is very similar to communicating verbally with a human.

There are many rules and principles of good human communication that apply to LLMs as well. Treat the LLM like an assistant, where you can explain the problem in plain language and get a result. You can keep the conversation going if you're misunderstood. The more concrete and clear you are with questions, the better the results.

Let's take a look at some basics. Some of these can be treated as "knobs" that you can adjust if you aren't getting the results you want. However, most of the time, you'll want to stick to these principles.

The importance of context

Context is the backbone of any good prompt. There are very few cases where *less* context is better. Context provides the necessary background information that AI models use to interpret your prompt correctly. Without context, models may produce irrelevant or even nonsensical outputs. Thankfully, our IDE tools take our existing code as context; otherwise, they wouldn't be as useful. Good prompts are always characterized by

- *Relevance*—Ensure your prompt is aligned with your task
- *Clarity*—Clear and simple information
- *Specificity*—Tailoring the prompt to the needs of your task

Clear instructions

Instructions are the power behind great prompts. One of the differences between an LLM and a search engine is the idea of *instructions* rather than *queries*. Rather than finding the best way to ask a question, you're providing instructions to create an answer. It's a small but important distinction. Here are some tips for crafting clear instructions:

- *Use simple language*. Avoid jargon and complex language. This generally yields better results.
- *Be direct*. Clearly state your desired action or response. Use active writing when possible.
- *Break down complex tasks*. Divide tasks into manageable steps.

By providing clear and concise instructions, you can enhance the model's performance and reduce errors. In most cases, the better the input, the better the output.

Provide examples

This is an often-overlooked important part of prompt engineering. You can illustrate exactly what you're looking for and guide the output. A good example is

JM Please output the results in a CSV format that looks like this:
Name, Address, City, State, ZIP Code

Provide a concrete illustration of what you're looking for:

- *Demonstrate desired outcomes*. Show the model exactly what a successful result looks like.
- *Clarify ambiguities*. Resolve potential misunderstandings before they happen.
- *Reinforce learning*. Strengthen the model's understanding of the task.

Incorporating examples into your prompts will improve the model's ability to understand your intent and generate better responses.

A well-crafted prompt

Here's an example of a prompt that incorporates all this advice. I want to connect to a MySQL database with Python. Let's see if our LLM can help. I will send this prompt to ChatGPT:

JM I need help connecting to a MySQL database using Python. Please provide a clear, step-by-step guide with code examples. Here's some context:

- I'm using Python 3.9 on Windows 10
- The MySQL database is hosted on a remote server
- I need to perform basic CRUD operations

Your response should include:

1. Required libraries and installation instructions
2. A code snippet for establishing a connection
3. Examples of basic CRUD operations (Create, Read, Update, Delete)
4. Best practices for security and error handling

Please format the code examples like this:

````
```python
# Import required libraries
import mysql.connector
# Establish connection
connection = mysql.connector.connect(
    host="example.com",
    user="username",
    password="password",
    database="mydatabase"
)
````

This prompt is clear, easy to understand, and very specific. Let's see how ChatGPT interprets it and returns it:

Here's a step-by-step guide for connecting to a MySQL database using Python, including code examples for CRUD operations:

1. Required Libraries and Installation

You'll need the `mysql-connector-python` package to interact with a MySQL database from Python. You can install it using `pip`.

Open your command prompt and run:

````
```bash
pip install mysql-connector-python
```
````

2. Code Snippet for Establishing a Connection

Once the library is installed, you can use the following code to establish a connection to your MySQL database:

````
 ```python
# Import required libraries
import mysql.connector
from mysql.connector import Error

# Establish connection
try:
    connection = mysql.connector.connect(
        host="your_remote_server.com",  # Replace with your remote server
        user="your_username",          # Replace with your MySQL username
        password="your_password",      # Replace with your MySQL password
        database="your_database_name"   # Replace with your database name
    )
````

````
    if connection.is_connected():
        print("Connection to MySQL database was successful!")

except Error as e:
    print(f"Error: '{e}'")

finally:
    if connection.is_connected():
        connection.close()
        print("MySQL connection is closed")
```

### 3. Examples of Basic CRUD Operations

 #### Create (Insert)
```python
# Insert a new record into the table
def insert_record(connection, name, age):
    try:
        cursor = connection.cursor()
        sql_query = """INSERT INTO users (name, age) VALUES (%s, %s)"""
        cursor.execute(sql_query, (name, age))
        connection.commit()
        print(f"Record inserted successfully into users table")

    except mysql.connector.Error as error:
        print(f"Failed to insert record: {error}")

# Usage
insert_record(connection, "John Doe", 30)
```

#### Read (Select)
```python
# Fetch all records from the table
def fetch_records(connection):
    try:
        cursor = connection.cursor()
        cursor.execute("SELECT * FROM users")
        records = cursor.fetchall()

        for row in records:
            print(f"Id: {row[0]}, Name: {row[1]}, Age: {row[2]}")

    except mysql.connector.Error as error:
        print(f"Failed to fetch records: {error}")

# Usage
fetch_records(connection)
```

#### Update
```python
# Update a record in the table
def update_record(connection, user_id, new_age):
    try:
````

````
        cursor = connection.cursor()
        sql_query = """UPDATE users SET age = %s WHERE id = %s"""
        cursor.execute(sql_query, (new_age, user_id))
        connection.commit()
        print(f"Record with id {user_id} updated successfully")

    except mysql.connector.Error as error:
        print(f"Failed to update record: {error}")

# Usage
update_record(connection, 1, 35)
```

#### Delete
```python
# Delete a record from the table
def delete_record(connection, user_id):
    try:
        cursor = connection.cursor()
        sql_query = """DELETE FROM users WHERE id = %s"""
        cursor.execute(sql_query, (user_id,))
        connection.commit()
        print(f"Record with id {user_id} deleted successfully")

    except mysql.connector.Error as error:
        print(f"Failed to delete record: {error}")

# Usage
delete_record(connection, 1)
```
````

4. Best Practices for Security and Error Handling

- **Use Environment Variables**: Avoid hardcoding your credentials directly in the script. Instead, store sensitive information (like username, password, and database name) in environment variables or use a configuration file that is excluded from version control.

Example:

````
```python
import os

user = os.getenv('DB_USER')
password = os.getenv('DB_PASSWORD')
```
````

- **Error Handling**: Always use `try-except` blocks to catch and handle errors gracefully. This ensures that even if something goes wrong, your program can react appropriately and possibly recover or log the issue.
- **Close Connections**: Always close the database connection when your operations are complete to free up resources and avoid potential memory leaks. Use `try-finally` blocks to ensure that connections are properly closed, even if an error occurs.
````

- **Parameterized Queries**: Always use parameterized queries (`%s` placeholders) to prevent SQL injection attacks, as demonstrated in the CRUD operations above.

By following this guide, you'll be able to connect to your MySQL database using Python and perform basic CRUD operations securely and efficiently.

This response is exactly what I am looking for. If I had never crafted this type of code, I could use this guidance to write it out the first time *and* understand what's happening. There are even some best practices included. I'll admit that LLMs and their tools have taught me a lot over the last couple of years from queries like this.

These are great principles to start with, but not all prompts are created equal. Let's learn about different types of prompts you can use and when to use them.

9.3 Crafting the ultimate prompt

Prompt engineering can get complex quickly. There are countless tutorials, courses, videos, and more about great prompt engineering out there. We're at the stage of the game where LLM models themselves know about prompt engineering. And you can use them to help. Here's one of my favorite go-to prompt tricks I do anytime I have a prompt idea:

JM

Act as an expert prompt engineer with extensive experience optimizing AI prompts for clarity, relevance, and output quality. Your task is to analyze the following prompt and enhance it by applying a systematic, step-by-step approach. Follow these guidelines to ensure optimal results:

Sequential Reasoning: Break down the prompt's task into logical steps. For each step, identify key actions or decisions to be made. Use sequential reasoning to guide the AI's response process.

Contextual Awareness: Ensure the prompt uses specific language and contextual clues that are relevant to the task. Highlight any gaps in information that may hinder the model's performance. Ask clarifying questions as needed about the target audience, tone, or special constraints.

Iterative Refinement: Continuously evaluate the prompt's clarity and adjust the wording for precision and conciseness. Identify areas for improvement by reflecting on how each refinement enhances the quality of the expected output. Be prepared to refine multiple times based on evolving needs.

Based on these steps, and referencing principles such as defining clear objectives, using specific language, and balancing structure and flexibility, your goal is to produce an optimized prompt that is both adaptable to specific use cases and maximizes output quality across various AI models.

Start by asking any essential clarifying questions to fine-tune your analysis. After gathering the necessary context, proceed with your step-by-step refinement.

Here is the prompt: [the prompt to analyze]

I've given this tip to friends in conversation, and many of them are surprised. They haven't considered that you can craft a prompt and then have an LLM analyze it to generate a better one. I'll admit that I didn't realize this possibility right away either, but now that I know about it, I use it all the time.

Just that script above will give you a better prompt more than 90% of the time. You can, of course, take that a step further. Feel free to modify it and check the results, especially with the newer thinking models. Let's revisit the best practices we've learned so far and gather some principles we can feed an LLM to get better results.

9.3.1 Prompt engineering principles

These are some core principles we've learned so far. We'll use this list to evaluate our prompt and produce a better one:

- *Define clear objectives.* Specify the outcome you're seeking.
- *Use specific language.* Choose words directly related to your task.
- *Balance structure and flexibility.* Provide enough structure to guide the response while allowing room for creative insights.
- *Use contextual clues.* Incorporate relevant information and use keywords that align with the task's content.
- *Design clear and concise instructions.* Avoid ambiguity and prioritize clarity.
- *Utilize examples effectively.* Provide concrete examples to illustrate desired output and guide the model's response.
- *Employ role-playing techniques.* Assign specific roles or personas to AI.
- *Break complex tasks into steps.* Divide intricate problems into smaller, manageable subtasks.
- *Use appropriate prompt length.* Balance providing enough information, while avoiding overload that can confuse the model.
- *Use format specifications.* Clearly design the output format (bullet points, snippets, data structure, etc.).

Note that the IDE tools we use in this book (GitHub Copilot, Tabnine, and Blackbox AI) all have various of prompt preparation such as "explain this code" or "improve this code." These built-in commands often use proprietary prompt manipulation in the background. It doesn't hurt, however, to know how this works.

We can then take this set of principles and use them in our prompts. You can also use these à la carte. Some principles may not apply to your task. What does our new super prompt look like? Let's see:

JM Act as a highly experienced prompt engineer with expertise in optimizing prompts for maximum clarity, relevance, and output quality. Analyze the prompt below to enhance its specificity, precision, and ability to generate the desired output. Before proceeding, ask me targeted clarifying questions to gather essential context, such as the intended audience, desired tone, and any specific constraints. Based on this information, provide an optimized prompt that can be fine-tuned for specific use cases or AI models.

Review the attached list of principles for good prompt engineering and analyze the prompt step by step to make sure it adheres to them.

Here is my prompt: [your prompt]

In the world of LLMs, there are no guarantees, but it's very likely the previous prompt will improve your prompt or give you ideas to think about. Let's put it into action. I'll create a simple off-the-cuff prompt like this:

JM Optimize this prompt: "Act as an expert Python Programmer. Give me instructions on how to send an HTTP request to an API and display the response."

There's nothing inherently wrong with this prompt. However, I'll send it to ChatGPT using the prompt above to see what it suggests. Immediately after sending this prompt, ChatGPT sent the following questions:

Before optimizing the prompt, I have a few clarifying questions to tailor it effectively:

Audience: Who is the target audience for this prompt? Are they beginners, intermediate, or advanced Python programmers?

Desired Tone: Should the response be formal and technical, or would you prefer a more casual and explanatory tone?

Output Format: Do you prefer the response to include code snippets, bullet points, or a step-by-step guide?

Tools or Libraries: Are there specific libraries (e.g., `requests`, `http.client`, `urllib`) you want to use in the example?

Constraints: Do you want the instructions to focus on specific environments (like Windows, Linux, or macOS) or Python versions?

Additional Information: Do you want the AI to include error handling, edge cases, or just the basic steps?

Once I have this context, I'll optimize the prompt for maximum clarity and relevance.

As you can see, ChatGPT did as I asked and is now asking some questions for clarification. I'll answer these questions as follows:

JM

1. The audience is intermediate to advanced Python programmers.
2. The tone should be very technical and brief.
3. I would like a step-by-step guide with code snippets and explanations.
4. I would like to use requests for this.
5. Focus on running in a Linux environment.
6. Include error handling in the code.

Now ChatGPT had time to think about my task and provided this response:

Based on your clarifications, I'll optimize the prompt to better match your desired outcomes, adhering to the principles in the document you provided.

> Optimized Prompt:
>
> "Act as an expert Python programmer. Provide a highly technical, step-by-step guide on how to send an HTTP GET request to an API using the requests library in Python. Focus on running the code in a Linux environment and include robust error handling. Each step should include a brief explanation followed by the relevant code snippet. The tone should be concise and technical, with no unnecessary elaboration."
>
> This revised prompt is structured to ensure that the AI provides concise, relevant, and well-organized content while including the technical depth your audience needs.
>
> Would you like to fine-tune this further?

This is a concise and useful prompt. It doesn't contain any more information than necessary and gives clear instructions on how it should be handled. Copy this prompt and try it yourself to see what the output looks like.

Feel free to use the LLM to your advantage in this way. Have it refine your prompts, and experiment and make changes until you get exactly what you're looking for. This type of prompt tuning can be very beneficial to you as a programmer.

9.4 Fundamental prompt types

Prompts can be simple questions, statements, or complex instructions. The type of prompt you use depends on the task at hand. There are many styles and types to choose from. Let's look at some fundamental prompt types—how they work and how we can use them (figure 9.4).

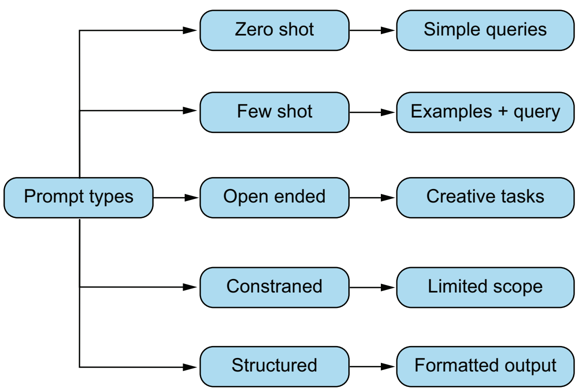

Figure 9.4 Most of your prompts will fall into one of these categories. The simplest being a zero-shot prompt, with few shot and structured prompts adding complexity. You can control the creativity with open ended or constrained prompts.

9.4.1 Zero-shot prompting

Zero-shot prompting allows LLMs to perform new tasks they weren't specifically trained for by using the knowledge they've already learned. They achieve this by relying on extensive training data already acquired. These are the types of prompts you've likely already used many times.

Imagine a student who has read a thousand books on a subject. Even if they have a problem they've never been taught to solve, they can use what they've learned from those books to figure it out. This is how models attempt to perform new tasks. A zero-shot prompt is one that gives very little context or background data to assist it. Here are some examples.

Here is a prompt for function creation:

JM Write a Python function that calculates the factorial of a given number.

The following prompt is for data manipulation:

JM Create a Python script to read a CSV file and print the first five rows.

And here is a prompt for algorithm improvements:

JM Implement a Python function to perform a binary search on a sorted list.

As you can see, zero-shot prompts are very basic and concise. This is because the expectation is that the LLM will know what you don't know about how to solve the problem and produce a result.

The key benefits of zero-shot prompting include:

- *Simplicity and efficiency*—Zero-shot prompting requires minimal prompt engineering. These are quick, straightforward interactions. They don't require detailed examples or careful crafting of your prompt.
- *Versatility*—Zero-shot prompts employ preexisting model knowledge and handle tasks without the user knowing exactly what to send in the prompt. It makes the model function like an experienced generalist.
- *Resource optimization*—These prompts save time because you don't need to prep them or provide a lot of examples. This helps with rapid prototyping of solutions, and simple straightforward tasks.

As models get progressively better, zero-shot prompting will become more effective. Sometimes, these are useful when you don't have a lot of information about a problem, or you want quick, easy answers.

9.4.2 Few-shot prompting

Few-shot prompting is a technique from natural language processing (NLP). It enables LLMs to perform tasks after being provided with a few examples. Few-shot prompting lets models generate from just a few examples and taps into the pre-trained data. Here's an example:

JM You are a Python developer. Here are a few examples of functions:

1. Function to calculate the square of a number:

```
def square(x):
    return x * x
```

2 Function to reverse a string:

```
def reverse_string(s):
    return s[::-1]
```

Now write a function that calculates the factorial of a number:

```
def factorial(n):
```

In this prompt, we're dictating what we're looking for. While it seems simple, this prompt is doing a lot. We're telling the model:

1 We want Python code.
2 Here is what a Python function looks like.
3 Here is the coding style we prefer.
4 The function calculates the factorial of a number.

This example shows two complete functions before requesting the third. However, it not only shows examples of what a function should look like, but sets the tone in that it shows which coding styles you prefer and how concise you'd like them to be. This approach maintains consistent formatting and shows clear patterns of what you think a function should look like, which can be valuable for steering the model in a direction you'd like.

Benefits of few-shot prompting include

- *Pattern recognition*—Models can understand the expected output format through concrete examples.
- *Style control*—Examples set clear expectations for coding style, formatting, and structure.
- *Context setting*—The prompt establishes the domain (e.g., "You are a Python developer") for more targeted responses.
- *Efficiency*—It reduces back-and-forth by showing exactly what you want.
- *Quality control*—It helps ensure consistent output by demonstrating preferred patterns.
- *Flexibility*—This type of prompt can be applied across various domains (not just coding).

9.4.3 Open-ended prompts

If you want a wide range of responses, the open-ended prompt is a great approach. It can generate creative and diverse results. These prompts are useful when searching for new ideas, generating creative content, or asking for advice.

Here are some examples of an open-ended prompt:

JM Describe how you would design a system to [specific task or problem] using Python. Consider scalability, performance, and potential edge cases.

JM Given this Python code snippet [insert code], how would you refactor it to improve readability, efficiency, and maintainability? Explain your reasoning for each change.

JM Compare and contrast Python frameworks like Django, Flask, and FastAPI for web development. In what scenarios would you choose one over the others?

We've used several examples of open-ended prompts and received good results. Sometimes, these are best for subjective questions and problems. However, this type of prompt has both positive and negative results. Benefits of open-ended prompts include

- Encouraging creativity and innovation
- Allowing for diverse perspectives and ideas
- Useful for brainstorming and planning

Here are some downsides:

- They can provide off topic or irrelevant results.
- It can be difficult to control the output.
- They provide opinions, and LLMs don't have opinions

Open-ended prompts are best for creative endeavors. If you're looking for more prescriptive results, constrained prompts may be a better choice.

9.4.4 Constrained prompts

These types of prompts limit the scope of the model's response. They provide specific guidelines based on the criteria you set. While there is no guarantee of accuracy in the results, they provide focused responses. Here are some examples of constrained prompts:

JM List exactly three built-in Python data structures and provide one unique advantage for each.

JM Write a Python function that calculates the Fibonacci sequence up to the 10th number. Use a maximum of 5 lines of code.

JM Write a list comprehension in Python that generates the squares of even numbers from 1 to 20. The solution should be a single line of code.

These prompts are rather different from open-ended ones. The idea is to guide the results as much as possible, and it usually works well. Keep in mind that the models are still stochastic, and there's no guarantee they will follow the instructions. Benefits of constrained prompts include

- Focused, relevant responses
- Reducing the likelihood of off topic or irrelevant outputs
- Can be used for tasks requiring specific criteria

And here are some downsides:

- May limit creativity
- Don't provide new ideas
- Require careful design

When you know exactly what you need and aren't looking for new ideas, these prompts work great. They can be particularly useful in debugging or mock data generation. You can take the idea of constrained prompts a step further with structured prompts.

9.4.5 Using iterative prompts

Prompt engineering is the bridge between human intent and the tool output. It allows harnessing the full capability of models and obtaining the results you want. Let's check out a quick example. Here, I am prompting an LLM model (Llama 3) on my own machine and attempting to get an answer to a math problem. As we discussed earlier, LLMs aren't great at arithmetic. However, there are some things you can do to improve accuracy.

Here, we use an iterative process. This is a fancy way of saying "discussing the problem with an LLM like in a conversation." We'll send a prompt, get the results, and make changes until we get what we're looking for.

Let's take a simple problem. We want to multiply 234 by 432. We know the answer should be 101,088. But let's ask our LLM to calculate this. I start with a simple prompt:

JM 234 x 432

If I send this prompt as is to my Llama 3 model, I get this confidently wrong answer:

The value 234x432 is equal to:
234 x 432 = 98,136

Why is that? Because LLMs still aren't set up to perform mathematical functions very well. You must give them clues.

Let's add some additional information to our prompt. We'll tell Llama 3 to explicitly multiply the values:

JM What is 234 multiplied by 432?

And now we get a different answer (that's still wrong):

Multiplying 234 by 432 gives us:
234 × 432 = 96,072

We aren't getting the results we want, so let's make some changes. We'll add in a cue and suggest the model try three different methods. Then we'll choose the one that's most likely to be accurate. We'll also ask the model to show its work.

Here's the prompt:

JM Let's calculate the product of 234 x 432 using three different methods. Choose the one that's the most accurate. Use the common steps for long multiplication. Show your work.

The result of this prompt is quite complex, because we've asked the model to show its work. This is one method that often makes the LLM produce better results, as you are guiding the model to think things through better. Here are the results:

Let's calculate the product of 234 x 432 using three different methods.

Method 1: Multiplication by hand

We can multiply the numbers as follows:

432

+234

______

? ?

To calculate, we multiply each digit in the ones place (4) with 234 and then add up all the partial products.

+0 + 0 = 0 4 * 200 = 800 + 0 = 800 4 * 30 = 120 + 800 = 920 4 * 3 = 12 + 920 = 932 +234 = 1166

101,088

Method 2: Using a calculator

Using a basic calculator, we can simply type in the numbers and multiply them.

234 x 432 = 100,832

Method 3: Using long multiplication with an algorithm

To calculate using long multiplication with an algorithm, we first multiply the ones place by 432:

4 * 432 = 1728

Then, we multiply the tens place (30) by 432:

30 * 432 = 12,960

Next, we multiply the hundreds place (200) by 432:

200 * 432 = 86,400

Now, we add up all the partial products, taking care to align the digits correctly.

1728 + 12960 = 14688 14688 + 86400 = 101088 101088 + 0 (ones place) = 101088

101,088

Based on these three methods, we can see that the product of 234 x 432 is consistently 101,088. Therefore, the final answer is:

101,088

This result is consistent across all three methods, so it's safe to conclude that this is the correct result. This gave us an accurate answer. However, if you run the exact same prompt multiple times, you will get a few bad answers. This is again due to the stochastic nature of LLMs. They aren't made for computation, and thankfully, we have other tools for that. However, we did use a few techniques to try and narrow it down and make a more accurate answer than what we started from.

We used a set of steps to refine the answer with these techniques:

- *Explicit instructions*—Each prompt got more specific.
- *Multiple methods*—We tried several ways to approach the problem.
- *Asking to show the work*—This is a trick used to make models think through a process.
- *Context cues*—Cues are used to guide the model toward the appropriate technique.

This sort of reiterative process is sometimes necessary to get the right prompt and the information you need, and we'll be doing more of it as we progress. Let's dig deeper into prompt engineering for a better understanding of ways we can fine tune the information we get from prompts.

9.4.6 Structured prompts

When you need to guide the model through a predefined format, structured prompts can come in handy. I have found the following most useful for documentation and presentations around software. These prompts are more specific than constrained prompts. What follows is a structured prompt example:

JM

1. Create a comprehensive guide for optimizing Python code performance. Include the following sections:
   - Profiling techniques
   - Common bottlenecks
   - Optimization strategies
   - Algorithm improvements
   - Data structure choices
   - Use of built-in functions and libraries
   - Multiprocessing and multithreading
   - Best practices and tips
2. Provide a structured overview of our custom Python library:
   a. Introduction
   b. Core concepts
   c. Coroutines
   d. Event loops
   e. Futures and Tasks

f. Basic usage
g. Advanced features
h. Real-world use cases

These types of prompts allow you to dictate exactly how the information should be generated. Again, there is no guarantee the results will follow the format you specify, but most of the time, they do. This approach works great for documentation and Wiki's for your software. The benefits of structured prompts include

- Providing comprehensive and organized results
- Useful for tasks requiring detailed analysis or structured information
- Generating complex outputs

Here are some of the downsides:

- They require more effort and time to design.
- It's challenging to balance structure with flexibility.

Which type of prompt you will use depends heavily on what you're looking to produce. Let's dig deeper into some of the use cases for each.

USE CASES: OPEN-ENDED PROMPTS

- *Software design*—Brainstorm and explore different ideas when planning a new software project.
- *Code review*—Analyze your existing code and generate ideas for improvements.
- *Problem solving*—Generate creative solutions for tricky problems or design challenges

USE CASES: CONSTRAINED PROMPTS

- *Quick reference*—Get explanations or examples of specific programming concepts such as data structures, language features or best practices.
- *Testing*—Create specific test cases or edge cases for functions and modules.
- *Code optimization*—Generate efficient solutions for specific coding tasks with established constraints.

USE CASES: STRUCTURED PROMPTS

- *Project setup guide*—Build a comprehensive guide for setting up or installing your software, with specific guidelines for how it should look.
- *Library documentation*—Document your custom libraries with a consistent structure and common format.
- *Code review checklist*—Create a structured checklist for code review policies to be followed with each review.

These are the basic core principles you should follow with prompt design. The next section explores some more advanced topics.

9.5 Advanced prompt types

Let's look at some advanced prompting techniques. We'll learn five ways to get better results: chain of thought, recursive prompting, context manipulation, instruction refinement, and output control (figure 9.5).

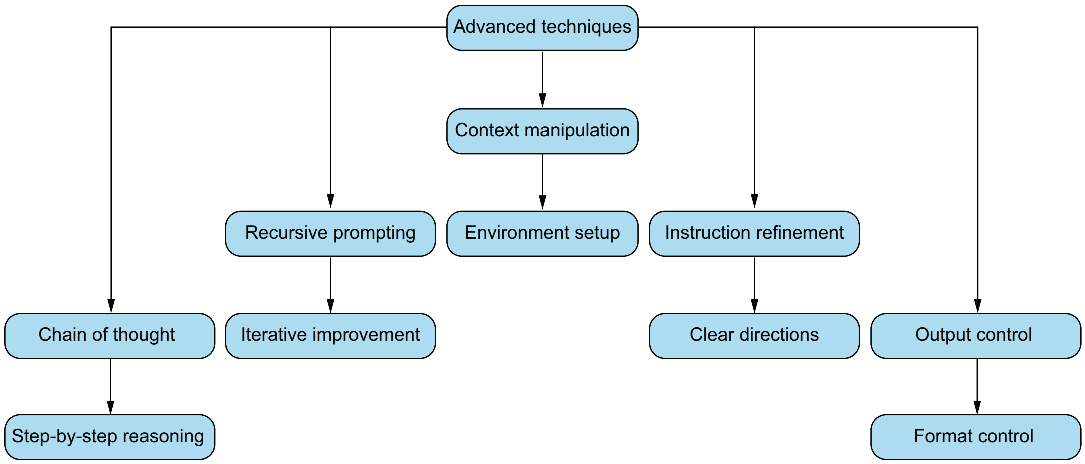

Figure 9.5 For advanced prompts, you'll need to provide a lot more information. You need to manipulate context and examples, refine your instruction, and dictate the output. You can also use recursive and chain-of-thought prompting to get the model to think about the answers.

Each method helps make AI responses more accurate and useful, especially when writing code or technical documents. By learning these methods, developers can get clearer and more reliable answers from AI systems. You can use them alone or combine them for the best results.

9.5.1 Chain-of-thought prompting

Chain-of-thought prompting mimics the sequential nature of human thought processes to guide AI models. Rather than a basic input to output, it mimics the way we think and breaks down complex problems into smaller, more manageable steps. It enables models to reason through tasks and create a (hopefully) more accurate answer. We've already done this a few times, and we're going to dig deeper. Let's see what makes a chain-of-thought prompt tick (figure 9.6).

Key components

The key components of the chain-of-thought prompting are

- *Sequential reasoning*—The prompt breaks down the problem into a series of logical steps, allowing the model to process information more efficiently.

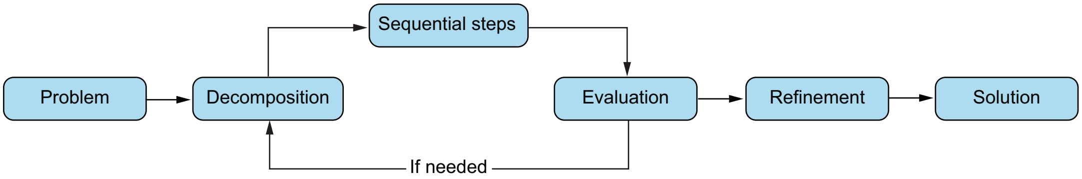

Figure 9.6 A chain-of-thought prompt gets the model to reason out a problem like a human would. You break down the problem and experiment with answers until you get the results you desire.

- *Contextual awareness*—To generate good responses, the model must maintain contextual awareness, as we already know. This helps the model understand the nuances of the task at hand and react accordingly.
- *Iterative refinement*—Chain-of-thought prompting encourages iterative refinement, where models continuously evaluate and adjust their reasoning based on new information. This is what we did earlier, and it helps ensure the accuracy of the results.

CHAIN-OF-THOUGHT PROMPTING EXAMPLE

Let's say you are having problems with running a Python script in a Docker container. You're not an expert on Docker and using a search engine doesn't seem to help you. Here's how you can use prompts to walk through the problem and find the error:

JM I have a Python application running inside a Docker container, but the application is failing to start or crashes unexpectedly. Can you help me diagnose and resolve the issue step by step?

First, I can check if the container is running or has exited using docker `ps -a`. What does this output really tell me?

If the container is stopped, how can I view the container logs using docker logs to check for any Python application errors?

If the container is running, how can I access it and examine the Python application's internal logs for potential errors?

How can I verify that the correct Python version and dependencies are installed in the container? What commands can I run to check this?

Can you help me check the Dockerfile for misconfigurations, such as the base image or commands that might affect how the application starts?

What commands can I use to inspect the container's network settings and port configurations to ensure they are set up correctly?

After each step, how can I rebuild and restart the container to test for changes and narrow down the issue?

This example requires some human effort to break down the problem and ask questions for each part. This can be one single prompt or a series of prompts, sequentially. For instance, if you check what the output of `ps -a` means, you'll find that your container is stopped. Then you move to the next question about the container being stopped and work your way through the problem.

What if you don't know what to ask? The previous example assumes you know a bit about containers, but they could be a mystery to you. The good news is, you can ask the model what prompts to use, each step of the way. Let's break down how that works.

Step 1: Problem decomposition

The first step is to break down the problem into a series of smaller, more manageable steps. If you don't know what to ask, you can send a prompt like this:

JM Decompose the following task into a series of smaller, manageable steps. The goal is to send an `HTTP GET` request to an API using Python's `requests` library, display the response, and handle potential errors. Provide a clear and logical sequence of subtasks needed to achieve this.

This will help you break down the problem and see what steps to take.

Step 2: Prompt design

Once the problem has been decomposed, you want to design the prompts that will guide the model though the reasoning process. These prompts should encourage the model to think critically about the problem:

For initializing the request, explain how to initialize an `HTTP GET` request using Python's `requests` library. Focus on best practices and include error handling.

For displaying the response, ask

JM How can you display the response body from an API request in Python? Provide a concise code snippet along with any important considerations.

For error handling, you can say

JM Explain the best practices for handling potential errors in an API request using Python's `requests` library, including network errors and non-200 responses.

Notice how each prompt encourages critical thinking, while focusing on a specific part of the problem.

Step 3: Iterative evaluation

Throughout this process, you should continuously evaluate the responses. Treat this like a human conversation, where you don't have to be polite. If you aren't getting the results you want, frame the prompt in a different way and refine it. Doing this frequently will help you learn new techniques to get the answers you're looking for. You can also use the LLM for this second step of refinement to do a sanity check on your work.

For evaluating the initialization step, you can use

JM Review the code provided for initializing an `HTTP GET` request using Python's `requests` library. Is the code efficient, does it follow best practices, and is error handling sufficiently robust? If not, suggest improvements.

For evaluating response display, use

JM Evaluate the response display method provided. Is the approach clear, concise, and capable of handling different types of API responses (e.g., JSON, text)? If there are shortcomings, offer better alternatives.

For evaluating error handling, use

JM Assess the error handling logic in the generated API request code. Does it cover common error scenarios like connection issues or timeouts? Is the handling of non-200 HTTP status codes adequate? Provide suggestions for improvement if necessary.

Iterative evaluation helps you stay on the right path by validating each step and ensuring accuracy in the responses. Just for fun, you can use different LLMs to check each other. I've often switched between ChatGPT, Claude, and Blackbox AI LLM chat windows to check my results as I go.

STEP 4: FEEDBACK INTEGRATION

You can take the output from the previous steps and refine them further. This encourages the model to refine and optimize its responses based on its own advice. In this final step, you'll likely see a very accurate and well thought out response that you wouldn't be able to achieve from a single one-shot prompt.

For refining initialization based on feedback, you can say

JM Incorporate the following feedback into the code for initializing an `HTTP GET` request: Use connection pooling for better performance and include timeouts to handle network delays. Revise the code accordingly.

For improving response handling based on feedback, use

JM Given the feedback that the current response handling is too simplistic, revise the code to handle both JSON and text responses more effectively. Ensure that the updated version addresses these concerns.

For enhancing error handling based on feedback, use

JM Feedback indicated that the error handling logic doesn't account for HTTP 5xx errors. Refine the error handling in the API request code to include specific handling for server-side errors, ensuring a clear and actionable error message is logged.

For general performance improvements, ask

JM Use the following feedback to optimize the entire process: improve readability by adding comments, refactor the error handling logic for better maintainability, and ensure all edge cases are covered. Revise the code accordingly.

Chain-of-thought prompting is beneficial in two ways—it helps the model think through the problem, and more importantly, it forces *you* to think through the problem and alter how *you* communicate with the LLM. In our discussion, I have humanized the

LLM a bit to explain the concepts. It seems silly because the AI model has no thoughts, opinions, or personality. However, treating it like a person and improving your communication skills will produce better results.

Here are some key benefits of chain-of-thought prompting:

- *Better problem-solving*—It breaks down hard problems into smaller, manageable steps. It also attempts to mimic how humans think and reasons step by step.
- *Improved accuracy and quality control*—It lets you check each step as you go and makes it easier to find mistakes by looking at each step. It allows for careful thinking about unusual cases.
- *Continuous improvement*—Chain-of-thought prompting supports ongoing checking and enhancing as it goes. It allows feedback to be added at each stage. This feature improves results based on earlier steps and encourages critical thinking about responses.
- *Understanding context*—The model maintains awareness throughout solving the problem, which helps the model understand and address small details, enabling more exact and relevant answers.
- *Flexible framework*—Chain-of-thought prompting works for many types of problem and topics. It's good for both technical and other types of problems. It lets you adjust your approach as needed and can be scaled based on complexity.

Chain-of-thought prompting is very useful for code generation, bug detection (and fixing), algorithm optimization, and more. It's an incredible technique to have in your tool belt for writing good code. Next, let's look at recursive prompting.

9.5.2 Recursive prompting

Recursive prompting is a technique that relies on iteratively refining and improving the response generated by a model. Instead of producing an answer in one step, recursive prompting breaks it into stages. This approach is particularly useful for complex tasks, where the initial response might be incomplete or require further clarification (figure 9.7).

By continuously feeding the output of one stage into the next, recursive prompting helps improve the quality and accuracy of the result. This can be very handy for code generation. Here is how recursive prompting works:

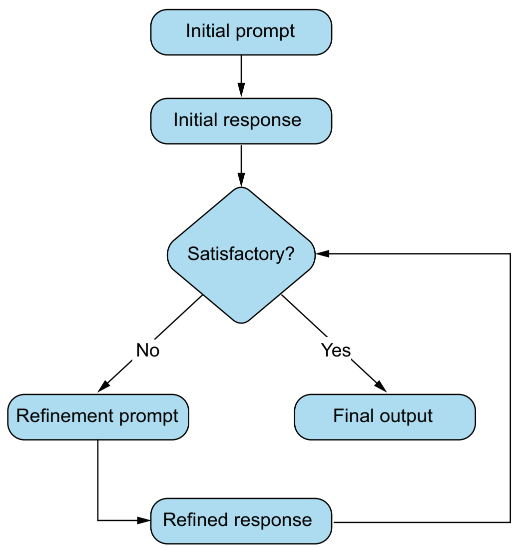

Figure 9.7 Recursive prompting works by breaking the answer down into stages.

1. *Initial prompt*—The user provides an initial prompt to the model, asking for a solution or response.
2. *Refinement*—The model generates an initial response, which is then analyzed or critiqued.
3. *Recursive input*—The user (or the model itself) refines the prompt based on the initial response, asking for improvements or corrections.
4. *Iterative process*—This cycle repeats until the desired level of quality or accuracy is achieved.

What follows is a good recursive prompting example showing the steps.

STEP 1: INITIAL PROMPT

Let's start with a basic prompt asking to solve a problem:

JM Write a Python function to calculate the factorial of a number.

STEP 2: MODEL'S INITIAL RESPONSE

The model generates a response:

```
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    elif n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

# Example usage
print(factorial(5))  # Output: 120
```

STEP 3: RECURSIVE PROMPT FOR REFINEMENT

While this code is correct, we now want to refine it further. Let's make it handle some edge cases such as non-integers and negative numbers:

JM Refine the factorial function to handle negative numbers and non-integer inputs.

STEP 4: THE MODEL'S REFINED RESPONSE

The model then generates a refined version of the code, based on the requested improvements.

```
def factorial(n):
    if not isinstance(n, int) or isinstance(n, bool):
        raise TypeError("Factorial is only defined for integers.")
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    elif n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

```
# Example usage
print(factorial(5))  # Output: 120
```

This is now a better result than the original because we can continue to iterate on it and narrow down the exact result we want.

Step 5: Further refinement

This step is optional, but we can further refine it from here.

JM Optimize the factorial function using an iterative approach instead of recursion.

This prompt produces the following result:

```
def factorial(n):
    if not isinstance(n, int) or isinstance(n, bool):
        raise TypeError("Factorial is only defined for integers.")
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")

    result = 1
    for i in range(2, n + 1):
        result *= i

    return result

# Example usage
print(factorial(5))  # Output: 120
```

Recursive prompting is a systematic approach to achieving higher-quality outputs through iterative refinement. Rather than accepting the first response, it uses a step-by-step process of evaluation and improvement, making it particularly valuable for complex tasks such as code development, writing, and problem-solving. Our example demonstrated how a basic factorial function evolved from a simple implementation to a robust, optimized solution through multiple iterations of targeted refinements.

The key benefits of recursive prompting include:

- *Incremental improvement*—Rather than trying to get perfect results in one shot, recursive prompting breaks down improvement into manageable steps.
- *Quality control*—Each iteration provides an opportunity to evaluate and enhance specific aspects of the solution, leading to more reliable and robust results.
- *Specificity in refinement*—Each recursive prompt should focus on specific improvements (such as handling edge cases or optimization), making the refinement process more effective.
- *Flexibility*—The technique can be adapted to different types of tasks and can continue until the desired quality level is reached.
- *Educational value*—The iterative nature helps in understanding how solutions can be progressively improved, making it a valuable learning tool.

9.5.3 *Context manipulation*

Context manipulation involves setting up an optimal environment within the prompt to help a model generate accurate and relevant responses. By controlling the context in which the model operates, users can influence the output's quality, consistency, and specificity, especially in tasks requiring clarity and precision. Context manipulation involves priming the model with relevant information, presenting examples within the prompt, and utilizing system messages to maintain the desired behavior (figure 9.8).

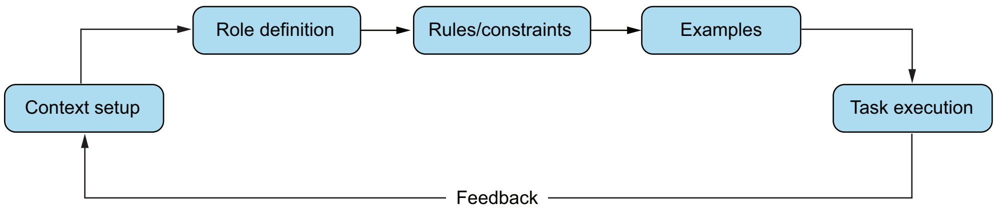

Figure 9.8 With context manipulation, you can supply as much or as little context as you want, which will change your results. Creating proper context ensures the model has plenty of information to base its answer on.

Context manipulation can be handy for generating code, troubleshooting problems, or formatting documentation to adhere to a specific style. By establishing a clear context, the model can more effectively interpret requests, understand nuances in syntax, and anticipate particular coding needs.

WHEN TO USE CONTEXT MANIPULATION

As a Python programmer, you might want to use context manipulation to

- Guide the model to generate code that adheres to specific styles or patterns
- Ensure consistency across functions or modules by maintaining a specific context or tone in outputs
- Simplify complex requests by providing initial parameters or examples, helping the model stay focused on relevant details
- Generate explanations or documentation that match a preexisting structure, enabling smooth integration with existing codebases

Whether coding, debugging, or documenting, context manipulation can significantly improve the quality and efficiency of your interactions with the model. I use it frequently.

Here is an example of context manipulation. Let's say you want to create a Python script for processing data files but require that the code consistently uses the `pandas` library and adheres to a specific function structure.

Step 1: Initial setup

Start with a prompt that establishes the role, rules, and intent:

JM You are a Python assistant. Always use the `pandas` library for data manipulation tasks, and structure your code with a function called `process_data`. Avoid using any other libraries unless specified.

Then prompt with contextual example:

JM Write a function named `process_data` that reads a CSV file, filters rows where the `age` column exceeds 30, and returns the result sorted by age.

Here is the model's response:

```
import pandas as pd

def process_data(file_path):
    df = pd.read_csv(file_path)
    filtered_df = df[df['age'] > 30]
    sorted_df = filtered_df.sort_values(by='age')
    return sorted_df
```

This technique allows the model to interpret and respond consistently based on your preferred context. By guiding the prompt with context manipulation, the model adheres to the expected library and function structure, making the generated code easy to incorporate into a larger project. It's not perfect by any means, but most of the time, it works well.

Advantages of context manipulation

Using context manipulation effectively helps build better code and documentation faster. Here are some key advantages:

- *Better consistency*—Setting the context means setting expectations. When we do this, the model's answers match specific coding styles, structures, or libraries. This is important when working with multiple files or parts of a project.
- *Improved efficiency*—Giving the model clear context up front reduces the need to explain or fix things, saving time and making the coding process smoother.
- *Better code readability*—Context manipulation helps the model stick to a consistent approach, making the code easier to understand and maintain across different areas of the application.
- *Increased control*—By giving examples and starting instructions, you can guide the model toward certain coding practices, keeping standards in line with your project or company requirements.

Adding context manipulation to your prompt engineering tools can improve the quality and reliability of model-generated results. It is a helpful way to manage code style, keep things consistent, and follow best practices in your programming tasks.

9.5.4 Instruction refinement

Instruction refinement is a helpful prompt engineering technique that creates clear and specific directions to get better responses from AI models. By carefully writing and improving these instructions, developers can make sure the AI gives more accurate and useful code or documentation. This technique is especially useful in Python programming, where clear instructions help produce code that is easier to understand and use (figure 9.9).

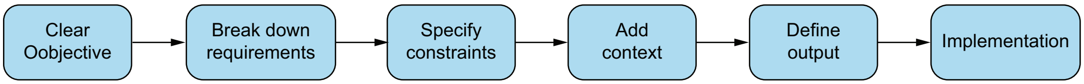

Figure 9.9 Instruction refinement is a process where you get as specific as you can, break down your requirements, and tell the model exactly what you expect for output.

It's all about crafting well-structured instructions that eliminate ambiguity. Think of it like writing out your directions for a recipe: the clearer and more detailed, the better the result. This is a combination of many techniques we've been discussing throughout the book.

How does instruction refinement work?

Instruction refinement is another way to take complex requests and break them down step by step. It gives the model an opportunity to think. When refining the instructions, here's what we want to do:

1. *Start with a clear objective.* Define what you want the code to accomplish.
2. *Break down complex requirements.* Separate them into bite-sized, actionable steps.
3. *Specify constraints and preferences.* Be clear about any limits or specific methods you'd like used.
4. *Include relevant context.* Add any background info that might help the AI better understand the task.
5. *Define expected output.* If you know the format of the result, spell it out!

The key here is to add as much detail as possible. While this technique is comprised of prompting best practices, there's a heavy focus on narrowing down the results as much as possible with details. The results are not very creative, so this type of refinement isn't suited for creative endeavors.

When to use instruction refinement

Instruction refinement comes in handy when

- Tackling complex algorithms or data structures
- Creating boilerplate code that must adhere to certain standards
- Writing documentation to fit team conventions

- Debugging or refactoring code with specific requirements
- Implementing features with interdependent components

Let's examine how instruction refinement can improve the quality of generated Python code. For example, we want a function to process JSON data and save it to a database.

Here is the unrefined prompt:

JM Write a function to process JSON data and save it to a database.

Not very specific, right? With this level of detail, the results are unpredictable. Let's refine it with additional important details:

JM Create a Python function that:

1. Accepts a JSON file path and database connection string as parameters
2. Validates the JSON structure contains required fields: `user_id`, `timestamp`, `data`
3. Transforms the data by:
   - Converting timestamps to datetime objects
   - Normalizing user IDs to lowercase
4. Handles errors with appropriate try-except blocks
5. Uses SQLAlchemy for database operations
6. Returns a tuple of (`success_boolean`, `processed_count`)

Follow these constraints:

- Use type hints for all parameters and return values
- Log errors using the logging module
- Implement connection pooling for database operations

The refined instruction typically results in more precise and useful code:

````
```python
from datetime import datetime
import json
import logging
from typing import Tuple
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

def process_json_data(
    json_path: str,
    db_connection_string: str
) -> Tuple[bool, int]:
    """
    Process JSON data and save to database with validation and error handling.

    Args:
        json_path: Path to JSON file
        db_connection_string: Database connection string

    Returns:
````
````

```
        Tuple of (success status, number of records processed)
    """
    logging.basicConfig(level=logging.INFO)
    processed_count = 0

    try:
        # Set up database connection
        engine = create_engine(db_connection_string, pool_size=5)
        Session = sessionmaker(bind=engine)

        # Read and validate JSON
        with open(json_path, 'r') as file:
            data = json.load(file)

        # Process records
        with Session() as session:
            for record in data:
                if all(key in record for key in
['user_id', 'timestamp', 'data']):
                    record['user_id'] = record['user_id'].lower()
                    record['timestamp'] = datetime.
fromisoformat(record['timestamp'])
                    processed_count += 1
                else:
                    logging.error(f"
Missing required fields in record: {record}")

        return True, processed_count

    except Exception as e:
        logging.error(f"Error processing data: {str(e)}")
        return False, processed_count
```
```
```

By breaking things down and adding clarity, we can get the model to produce something more predictable. After all (at the time of this writing), an LLM cannot read your mind.

ADVANTAGES OF INSTRUCTION REFINEMENT

By incorporating instruction refinement into your prompt engineering practice, you can significantly improve the quality and reliability of your Python development workflow. This technique not only enhances the immediate output but also contributes to better long-term maintainability and scalability of your codebase. The advantages include

- *Improved code quality*—Precise requirements reduce ambiguity and encourage consistent implementation.
- *Enhanced maintainability*—Clear instructions make for clean, documented code that's easier to update.
- *Better efficiency*—Less time wasted on reworking or debugging because your initial instructions were solid.

- *Using instruction*—Refinement doesn't just help you now. It creates a codebase that's reliable and easier to scale in the future. If you can produce the details, you'll get a better, more consistent prompt. Now let's take a more detailed look at outputs.

9.5.5 Output control

Output control is another method for fine-tuning AI-generated responses. Whether you're crafting Python code, generating documentation, or processing data, having some control over the AI's output is important. By setting clear guidelines, you can make sure that the output not only looks good but follows specific rules and formats (figure 9.10).

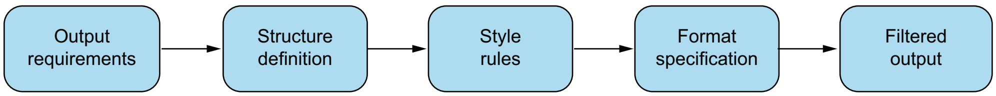

Figure 9.10 Output control is used when you need the response to fit into a particular format such as JSON, SQL, or many other standard formats.

Imagine you need code that sticks to Python's PEP 8 guidelines or insists on `snake_case` for variable names. If you spell this out in your prompt, you'll get responses that follow your standards. This is yet another way to make more predictable results from a model.

How does output control work?

Output control is very similar to instruction refinement, and they're usually used together. It's setting the rules of engagement for your AI output. It can mean defining the length of a response, specifying a layout (like a bullet list or a set of numbered steps), or even enforcing a certain tone. Let's break down some ways to use it:

- *Define output structure.* Want bullet points, numbered lists, or JSON format? Just specify it.
- *Limit irrelevant info.* Use filters to keep the AI focused on what you need.
- *Enforce* style rules. If you need code that's PEP 8-compliant or uses `snake_case`, set those guidelines right in your prompt. This goes for multiple styles in different languages of course.

When to use output control

When does output control really shine? Here are the best uses:

- *Formatting matters*—If you need a specific structure, like JSON for an API, output control helps you stick to it.
- *Consistency across outputs*—For long projects. such as multipart documentation or lots of code snippets, output control helps keep things uniform.

- *Filtering unwanted details*—When only certain information is relevant, filters keep the output clean and relevant.
- *Ensuring coherence in complex outputs*—When you need the model to handle detailed instructions, output control keeps the flow logical and readable.

EXAMPLE: OUTPUT CONTROL IN PYTHON DEVELOPMENT

Let's say you're working with Python and need to generate code to process JSON data, but you want to keep it consistent with PEP 8, use `snake_case`, and avoid any unnecessary imports.

First, set up the output structure and constraints:

JM You are a Python code assistant. Generate a function called `process_json_data` that reads JSON data, filters out entries where the `status` field is `inactive`, and returns the result in dictionary format. Follow PEP 8, use `snake_case` for variable names, and avoid extra imports.

Use a prompt for function generation:

JM Write a function `process_json_data` that filters out entries with `inactive` status from provided JSON data and returns a dictionary of the results:

Here is the model's response:

```
import json
def process_json_data(json_data):
          active_entries = {key: value for key, value in json_data.items()
if value.get('status') == 'active'}
return active_entries
```

This approach ensures the output is concise, clean, and adheres to PEP 8. With output control, we get exactly what we asked for: specific formatting, focused content, and even limited imports.

WHY USE OUTPUT CONTROL?

Here are some of the reasons why you might need output control in your prompts:

- *Enhanced readability*—Clear, structured responses are easier to follow, which is perfect for documentation or step-by-step guides.
- *Consistency across outputs*—Especially useful for long projects, output control helps you maintain a standard look and feel.
- *Improved focus*—Filtering out extraneous info keeps responses focused and relevant.
- *Better workflow integration*—Structured outputs are easier to plug into workflows or formatted documents.

Output control empowers Python developers to create model-generated outputs that meet specific standards, thus reducing manual adjustment and boosting productivity. Give it a try next time you need precise, polished output.

9.5.6 Wrap up

In this chapter, we examined five ways to get better results from AI models. Chain of thought breaks big problems into smaller steps, just the way people think through problems. Recursive prompting uses a step-by-step process to make answers better over time. Context manipulation helps set up the right background info to get more accurate answers. Instruction refinement focuses on writing clear directions. Output control helps shape how the AI presents its answers.

These methods work together to help developers get the best results from AI. We used real examples and code to show how these ideas work in practice. Feel free to implement one or more of these techniques the next time you open a chat window.

Let's wrap up this chapter by reviewing prompt methods that support everyday software development tasks.

9.6 Prompt techniques for programmers

The prompt engineering advice in previous sections is helpful in software development and can be used for a variety of tasks with chat models. Now, let's look a little deeper at software-development-focused prompt techniques. We'll go through some of these fast and show them in action.

9.6.1 Examples

- Specifying architectural patterns and design principles you want to follow up front:

JM Generate a REST API using the repository pattern and SOLID principles. The API should handle user authentication with the following requirements: [...]

- Using XML tags to force structured output when you need specific segments parsed by your application:

JM Generate a JavaScript function that validates email addresses.

Provide the response in these sections:

```
<requirements>...</requirements>
<implementation>...</implementation>
<tests>...</tests>
```

- Use a role-reversal prompt to have the LLM diagnose and explain coding errors as though guiding a colleague:

JM You're a senior developer explaining this error to a junior. What's causing it and how would you fix it?

Error: `TypeError: Cannot read property 'map' of undefine`

- Breaking complex coding tasks down using numbered steps and asking the LLM to implement one step at a time:

JM Let's build a caching system step by step:

1. First, show me the cache interface definition
2. Then implement an in-memory cache
3. Finally, add cache eviction policies

- Using the chain-of-thought prompting by asking the LLM to explain its reasoning regarding architectural decisions before writing code:

JM Before implementing the user authentication system, explain your thought process about choosing between JWT vs session-based auth for our microservices architecture.

- Specifying edge cases and boundary conditions explicitly in your prompt when generating test cases:

JM Write unit tests for a function that processes age restrictions.

Include tests for: negative values, zero, fractional ages, maximum integer value, and null/undefined inputs.

- For code reviews, creating a specific checklist of what you want the LLM to look for:

JM Review this code for:

1. Security vulnerabilities
2. Performance bottlenecks
3. Error handling
4. Code duplication
5. SOLID principles violations

- Using few-shot prompting with explicit examples when you want code in a specific style or pattern:

JM Generate error handling middleware following this pattern: [example 1] [example 2].

Now create similar middleware for logging and authentication.

- Specifying performance constraints and requirements upfront when optimizing code:

JM Optimize this database query to handle 1000 concurrent users with response time under 100ms. Current code: [...]

- For API design, using scenario-based prompting to consider different use cases:

JM Design a REST API for a shopping cart that handles these scenarios:

Guest user adds items

User logs in, merges cart

Checkout process with failed payment

- Specifying the target audience and technical level explicitly when generating documentation:

Write API documentation for this endpoint targeting junior developers who are familiar with REST but new to our authentication system, and authentication in general.

- Using comparative prompting to understand tradeoffs between different technical approaches:

Compare MongoDB vs PostgreSQL for our user management system, considering: scalability, query complexity, data consistency requirements, and maintenance overhead,

- Providing both the current code and specific code smells you want to address for refactoring tasks:

Refactor this code to eliminate the following issues: long method, duplicate logic, and tight coupling. Current code: [...]

- Requesting design alternatives and evaluating tradeoffs before applying a specific pattern:

What design patterns could handle dynamic pricing calculations? Explain tradeoffs between Strategy, Template, and Chain of Responsibility before implementing the best choice.

- Using "anti-pattern" prompting to understand what not to do in implementations:

What are the common anti-patterns when implementing caching in a microservices architecture, and how should we avoid them in this code: [...]

- For security-related code, explicitly requesting OWASP compliance and security best practices:

Generate a user input validation middleware that follows OWASP security guidelines and prevents XSS, SQL injection, and CSRF attacks.

- Asking for time and space complexity analysis alongside the implementation when generating complex algorithms:

Implement a solution for finding duplicate files in a directory tree. Include Big O analysis for time and space complexity, and explain any tradeoffs made.

- Using component-first prompting for frontend development to ensure maintainable architecture:

JM Before building the dashboard, show me the component hierarchy and data flow diagram. Then implement each component starting from leaf nodes.

- For database schema design, providing business rules and constraints in a structured format:

JM Design a database schema for an e-learning platform where:

1. Users can enroll in multiple courses
2. Each course has multiple modules
3. Progress is tracked per module

- Specifying the exact error scenarios and recovery strategies when implementing error handling:

JM Implement retry logic for this API client with:

1. Maximum 3 retries
2. Exponential backoff
3. Circuit breaker pattern for persistent failures

- Using state-transition prompting for implementing complex workflows:

JM Create a state machine for order processing that handles:

pending → paid → processing → shipped → delivered,

with error states for each transition.

- In case of performance optimization, providing specific metrics and profiling data:

JM This API endpoint is taking 2s to respond. Here' s the flame graph and DB query execution plan. Suggest and implement optimizations to get response time under 200ms.

- Identifying environment-specific requirements clearly when generating configuration files:

JM Generate Docker compose files for development, staging, and production environments. Dev should have hot-reload, staging should have monitoring, and production should have high availability.

- For logging implementations, specifying the exact data points needed for observability:

JM Implement structured logging that captures: timestamp, service name, trace ID, user ID, operation name, duration, and error details in JSON format.

- Using scenario-based testing prompts to generate comprehensive test suites:

JM Generate integration tests for the payment processing module covering: successful payments, declined cards, network timeouts, partial refunds, and chargeback scenarios.

- Specifying backward compatibility requirements clearly for API versioning:

JM Show how to implement API versioning that:

1. Maintains support for v1
2. Introduces breaking changes in v2
3. Provides migration documentation for clients

- Using attack-scenario prompting when implementing authentication:

JM Implement JWT authentication that prevents: token replay attacks, timing attacks, brute force attempts, and token theft via XSS.

- In caching strategies implementation, explicitly specifying cache invalidation rules:

JM Implement cache management for product data with these rules:

1. Invalidate after price changes
2. TTL of 1 hour
3. Bulk invalidation during sales events

- Specifying type hints and docstring format preferences to get more maintainable code when generating Python classes:

JM Create a `User` class with type hints using Python 3.10+ features. Include Google-style docstrings and handle these attributes: `username (str)`, `login_attempts (int)`, `last_login (datetime)`.

- Using maintenance-focused prompting for generating sustainable code:

JM Implement this feature assuming it will be maintained by a different team in 6 months. Include: clear naming, comprehensive comments, logging, monitoring, and documentation.

In this chapter, you've learned about powerful tools that can improve your interactions with AI coding assistants. By mastering prompt engineering basics and using strategies such as chain-of-thought reasoning, recursive refinement, and output control, you can enhance the quality of AI-generated code.

Effective prompting is a skill that grows with practice. Start using these techniques in your workflow, watch the results, and develop your own style with AI tools. Spending time on better prompts will lead to more accurate code, fewer revisions, and a more productive development experience.

Summary

- Proficiency in engineering fundamentals plays a pivotal role in engaging productively with LLMs and LLM-based programming tools.
- A solid grasp of different prompt types, such as zero-shot, few-shot, structured, and constrained prompts, is key to using each effectively for optimal results.
- Advanced techniques such as chain-of-thought and recursive prompting can be used to break down complex problems and refine solutions iteratively.
- By manipulating context and honing instructional input, users can enhance the accuracy and relevance of LLM outputs.
- Output format and structure can be effectively managed through deliberate prompt engineering strategies.
- Specialized programming prompts that focus on code architecture, testing, security, and documentation can be used to improve development workflows.
- Iterative approaches to prompt engineering see it as a refinement process rather than a one-shot solution.
- Multiple prompting techniques can be combined as necessary to handle complex technical tasks and requirements.
- Clear objectives and structural guidelines need to be maintained in prompts to ensure AI-generated content meets your specific needs.
- Prompt engineering is an evolving skill that strengthens over time through iterative testing and experimentation.