

Chapter 3. Prompt Engineering

Prompt engineering is a subfield of machine learning and *natural language processing*, which is the study of enabling computers to understand and interpret human language. The main goal is to figure out how to talk to *large language models*, sophisticated AI systems designed to process and generate human-like language responses, in just the right way so they generate the answer we're looking for.

Think of it like this: You know how when you ask someone for advice, you've got to give them a bit of context and be clear about what you need? It's like that with LLMs. You've got to craft your question or prompt carefully. Sometimes, you might even drop some hints or extra information in your question to make sure the LLM gets what you're asking.

This is not just about asking one-off questions either. Sometimes it's like having a whole conversation with the LLM, going back and forth, tweaking your questions until you get that golden nugget of information you need.

For instance, let's say you're using an AI-assisted programming tool to develop a web application. You start by asking how to create a simple user login system in JavaScript. The initial response might cover the basics, but then you realize you need more advanced features. So, you follow up with more specific prompts, asking about incorporating password encryption and connecting to a database securely. Each interaction with the AI hones its response, gradually shaping it to fit your project's specific needs.

Keep in mind that prompt engineering has become a red-hot job category. According to data from [Willis Towers Watson](#), the average yearly earnings of a prompt engineer hover around \$130,000, though this figure might be on the conservative side. To lure top talent, companies often sweeten the deal by offering enticing equity packages and bonuses.

In this chapter, we'll dive deep into the world of prompt engineering and unpack helpful strategies and tricks of the trade.

Art and Science

Prompt engineering is a mix of art and science. On one hand, you've got to choose the right words and tone to get the AI to respond the way you want. It's about guiding the conversation in a certain direction. It takes a bit of intuition and a creative touch to guide the conversation in a certain direction and refine your language, teasing out detailed and nuanced replies.

Yes, this can be tricky, especially for software developers. Normally, you follow a set of rules to write your code, and it either works or the compiler tells you what you did wrong. It's logical and predictable.

But prompt engineering? Not so much. It's more freeform and unpredictable.

Then again, there is also quite a bit of science to prompt engineering. You need to understand the nuts and bolts of how AI models work, as we discussed in [Chapter 2](#). Along with creativity, you need precision, predictability, and the ability to replicate your results. Often this means you've got to experiment, try out different prompts, analyze the results, and tweak things until you get the right response.

With prompt engineering, don't expect to find any magic solutions that work every time. Sure, there are plenty of courses, videos, and books that claim to have all the "secrets" of prompt engineering. But take them with a grain of salt, or you might be disappointed.

Plus, the world of AI and machine learning is always changing, with new models and techniques popping up all the time. So, the idea of having one definitive technique for prompt engineering? That's a moving target.

Challenges

Prompt engineering can be frustrating. Even the tiniest change in how you phrase your prompt can make a huge difference in what the LLM

spits out. This is because of the advanced technology under the hood, which is based on probabilistic frameworks.

Here are some of the challenges with prompt engineering:

Wordiness

LLMs can be chatterboxes. Give them a prompt, and they might just run with it, giving you a wordy response when all you wanted was a quick answer. They have a tendency to throw in a bunch of related ideas or facts, making the response longer than necessary. If you'd like an LLM to get straight to the point, just ask it to be "concise."

Non-transferability

This means that a prompt that works nicely with one LLM might not be as effective with another. In other words, if you're switching from ChatGPT to Gemini or GitHub Copilot, you might need to tweak your prompts due to the unique training, design, and specialization of each LLM. Different models are trained on different datasets and algorithms, leading to distinct understandings and interpretations of prompts.

Length sensitivity

LLMs can get overwhelmed by long prompts and start to overlook or misinterpret parts of your input. It's as if the LLM's attention span falters and its responses become somewhat distracted. This is why you should avoid providing detailed requirements in your prompts; keep a prompt to less than a page.

Ambiguity

If your prompt is unclear, the LLM might get confused and serve up responses that are way off base or just plain make-believe. Clarity is key.

Despite all this, there are ways to improve the results. And we'll cover these approaches in the rest of this chapter.

The Prompt

You can think of a prompt as having four main components, which you can see in [Figure 3-1](#).

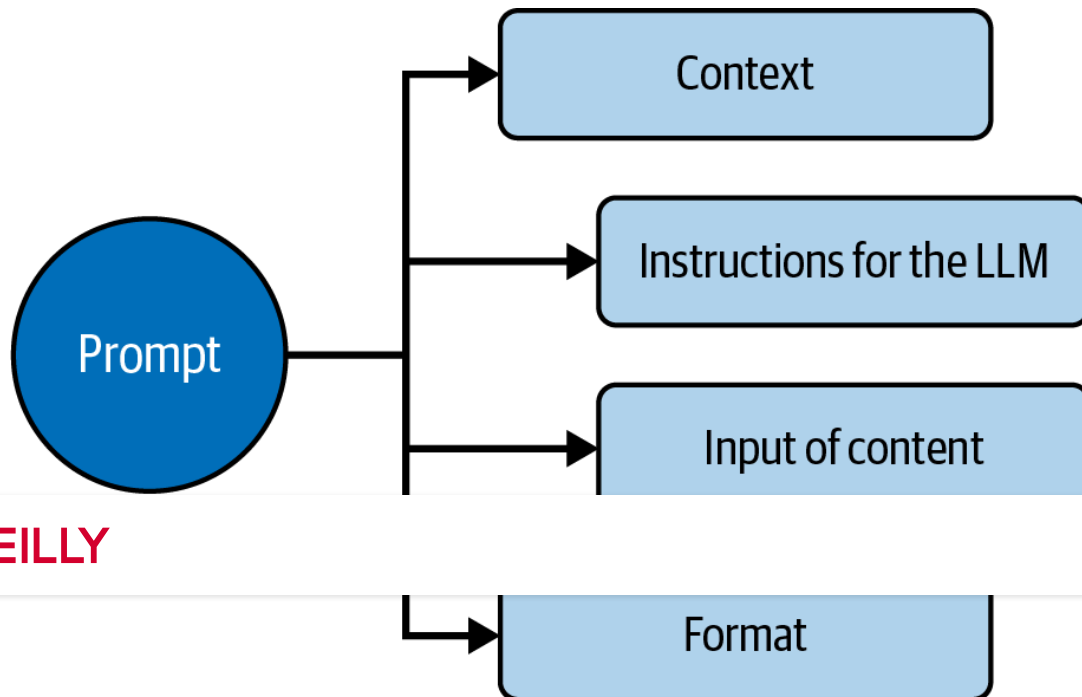


Figure 3-1. A prompt has four main components

First, the *context* specifies the persona or role for the LLM to take when providing a response. Next, there are the *instructions*, such as to summarize, translate, or classify. Then there is the *input of content* if you want the LLM to process information to create a better response. Finally, you can show how you want the output *formatted*.

Keep in mind that you do not need all of these components. In fact, you might need just one to get a good response. But as a general rule, it's better to provide the LLM with more concrete details.

Let's now look at each of the components.

Context

You'll often begin your prompt with a sentence or two that provide context. Often, you'll specify the role or persona you want the AI to take on when providing the response. This leads to responses that are not only more accurate but also contextually relevant, ensuring a more meaningful result.

For instance, if you want to debug a piece of code, you might use this as the context:

Prompt: You are an experienced software engineer specializing in debugging Java applications.

Or suppose you want to learn about optimization techniques for a particular algorithm. You could set the stage by stating:

Prompt: You are a senior software developer with expertise in algorithm optimization.

Adding context helps the LLM approach your prompt with the right mindset.

Instructions

Your prompt should include at least one clear instruction. There's nothing stopping you from adding more instructions, but you need to be careful. Loading up your prompt with a bunch of queries can throw the LLM for a loop and make it harder to get the answer you're looking for.

Let's break down why that happens. First off, when you have multiple instructions, things can get a bit fuzzy. If they're not clear or if they seem to clash with each other, the LLM might get confused about which one to focus on or how to balance them all out.

Next, having more instructions means more for the LLM to juggle. It's got to process and understand each part of your prompt and then figure out how to weave all the parts into a coherent response. That's a lot of mental gymnastics, and sometimes it can lead to mistakes or answers that are off.

And don't forget, LLMs go through instructions one at a time, in order. So, the way you line up those queries can influence how they're interpreted and what kind of answer you get back.

Given all this, a pro tip is to keep it simple. Instead of throwing a whole list of questions at the LLM all at once, try breaking them down into a series of smaller prompts. It's like having a back-and-forth chat instead of delivering a monologue.

There are also numerous types of instructions for a prompt. In the next few sections, we'll discuss some of the main instructions used in software development.

Summarization

Summarization can condense a longer piece of text into a shorter version while keeping the main ideas and points intact. This is useful for quickly getting a handle on lengthy documents. For a software developer, summarization can be an especially handy tool in the scenarios listed in [Table 3-1](#).

Table 3-1. Summarization prompts for coding tasks

Use case	Description	Example prompt
Code documentation	Provides a concise overview of extensive documentation highlighting key functionalities, dependencies, and structures.	“Summarize the main points of the following documentation to provide a quick overview of the codebase.”
Bug reports	Quickly identifies the main issues reported by users in numerous or lengthy bug reports.	“Summarize the common issues reported in the following bug reports to identify the main problems to be addressed.”
Research papers	Extracts succinct insights from lengthy research papers or technical articles to update the user on the latest research or technologies.	“Provide a summary of the key findings and technologies discussed in the following research paper.”
Change logs	Enables an understanding of the key changes in a new version of a software library or tool from lengthy change logs.	“Summarize the key changes in the following change log of version 1.1.2.”
Email threads	Extracts the key points of discussions or decisions from long email threads.	“Summarize the main points of discussion from the following email thread.”

Another type of summarization is *topic modeling*, in which a statistical model discovers the abstract “topics” that occur in a collection of documents. Here are some topic-modeling prompts for developers:

Prompt: Identify the main topics discussed in the following text:

{text}

Prompt: Extract the keywords from the following text to infer the main topics: {text}

Prompt: Suggest tags for the following text based on its content: {text}

Text Classification

Text classification involves giving a computer a bunch of text that it learns to tag with labels. A flavor of this is *sentiment analysis*, such as when you have a list of social media posts and the LLM figures out which have a positive or negative connotation. For developers, sentiment analysis can be a useful tool to gauge user feedback about an application.

Some sample prompts include:

Prompt: Can you analyze these customer reviews and tell me if the sentiment is generally positive, negative, or neutral? {text}

Prompt: Here's a thread from our user forum discussing the latest update. Could you summarize the overall sentiment for me? {text}

Prompt: I've compiled a list of feedback from our app store page. Can you categorize the comments by sentiment? {text}

Prompt: Evaluate the sentiment of these blog post comments regarding our product announcement. What's the consensus? {text}

Recommendation

You can instruct an LLM to provide recommendations. Developers can use such feedback to improve the caliber of responses for activities like squashing bugs, refining code, or using APIs more effectively.

Check out these example prompts you might use:

Prompt: The following code snippet is throwing a `NullPointerException` when I try to call `<Method()>`. Can you help identify the potential cause and suggest a fix?

Prompt: Here is a function I wrote to sort a list of integers. Can you recommend any optimizations to make it run faster or be more readable?

LLM recommendations can be a powerful accelerator for your work, greatly saving time and providing ideas you may not have thought about. This technique is particularly beneficial when dealing with intricate or nuanced tasks.

But there are downsides. One potential hitch is that the LLM might boil down the responses too much and miss the nuances. Also, keep in mind that the model's knowledge is frozen at a certain point in time, so it might not be up-to-date with the latest information or trends.

If anything, recommendations are a way to kick things off. But you'll want to dive in and do some more digging on your own to get the full picture.

Translation

Localization is essentially attuning the software to the linguistic and cultural norms of a specific area. It allows your software to speak the local lingo and understand regional quirks, an ability that is key to broadening your market and cultivating a closer connection with your audience. This can lead to a ripple effect of benefits: users are happier because the software feels tailor-made for them, and happy users can mean a healthier bottom line for your business.

In competitive markets, localization can give you an edge when alternatives fall short or simply don't exist. Plus, by aligning your software with the local ways, including compliance with regional regulations, you're not just making your software one option but often the only option for a market.

On the flip side, localization is not without its challenges. It can be both expensive and time intensive. It requires meticulous quality assurance to maintain the software's integrity in different languages. Additionally, software development doesn't stand still. It's a continuous cycle of updates

and new features, each of which may require its own set of localization efforts. This ongoing process adds layers of complexity and additional costs to the project.

This is where LLMs can come to the rescue. Advanced systems are capable of translating between numerous languages. They can serve as a powerful tool in a developer’s toolkit. [Table 3-2](#) shows some prompts you might use for localization.

Table 3-2. Examples of prompts for language translation

Task type	Description	Sample prompt
UI text translation	Translates buttons, menu items, error messages, dialog boxes, etc.	“Translate the following UI text into French: Save, Exit, File, Edit, Help.”
Documentation translation	Translates user guides, help files, and other documentation.	“Translate the following user manual paragraph into Spanish.”
Error message translation	Translates error messages that the software might generate.	“Translate the following error messages into German: File not found, Access denied, Network connection lost.”
Tooltip translation	Translates tooltips that provide additional information when a user hovers over an item.	“Translate the following tooltips into Japanese: Click to save, Click to open a new file, Click to print.”

Even so, it’s crucial to approach the multilingual capabilities of LLMs with a degree of caution. They aren’t foolproof. These models may sometimes miss the subtleties, idiomatic expressions, and cultural contexts unique to a language. The nuances of language are complex, and getting them right is about more than just direct translation—it’s about conveying the right meaning in the right way.

Handling specific terms or names can be tricky, especially when there isn’t a neat equivalent in another language. Then there’s the challenge of getting the tone and style right. It’s not just about the words but how you say them, and this can change a lot from one language or culture to the next.

Having a language specialist take a look at the output could save you some headaches down the line.

Input of Content

When crafting prompts, it's helpful to use special symbols like `###` or `"""` to clearly separate your instructions from the content or information you want the LLM to work on. These symbols act like boundaries or markers, making it clear where the instructions end and where the content begins.

Consider a scenario in which a software developer needs help summarizing key points from a lengthy piece of documentation regarding a new API they are integrating. Here's how you could structure the prompt:

Prompt: Extract the key implementation steps for the API from the text below:

Documentation: `"""`

{API documentation text here}

`"""`

Using the `"""` delimiters is a neat way to split the instruction from the API documentation text. It gives the LLM a clearer picture of what needs to be done and increases the chances of getting a crisp summary of the main steps. Plus, these delimiters tidy up the prompt, making it easier to read, which is a real lifesaver for longer or more complex text inputs.

Format

In your prompt, you can tell the LLM how to format the output. Here's an example:

Prompt: Create a Python function that takes a list of user objects (each object containing a user’s ID and name) and returns a JSON object that maps user IDs to names. Format the output as JSON.

ChatGPT:

```
{
  "code": "def map_user_ids_to_names(users):\n    user_map = {user['id']: user['name'] for user in users}\n    return user_map",
  "example_usage": {
    "input": [
      {"id": 1, "name": "Alice"},
      {"id": 2, "name": "Bob"},
      {"id": 3, "name": "Charlie"}
    ],
    "output": {
      "1": "Alice",
      "2": "Bob",
      "3": "Charlie"
    }
  }
}
```

There are other ways you can format the output. [Table 3-3](#) shows some options.

Table 3-3. Prompts for formatting output

Format type	Sample prompt
Table	“Create a table comparing the syntax, performance, and use cases of Python, Java, and C++.”
List	“List the steps to troubleshoot a slow-loading web page.”
Markdown/HTML	“Explain the differences between GET and POST HTTP methods in Markdown.”
Text hierarchy	“Provide a structured outline of the software development life cycle (SDLC), including its phases and key activities in each phase.”
LaTeX formatting	“Express the time complexity of the binary search algorithm in LaTeX notation.”

With a prompt, you can also specify the length of the response. You could guide the LLM with an instruction such as “Provide a brief summary” or “Write a detailed explanation.” Or you could be more specific, such as by saying that the response should be no more than 300 words. The LLM may exceed the word limit you provide, but it will at least be in the general vicinity.

Best Practices

We’ll next take a look at some of the best practices for cooking up prompts that will help get the answers you want. But don’t take these as gospel. These suggestions are more like general advice—which can be somewhat subjective—than hard-and-fast rules. As you spend more time chatting with LLMs, you’ll probably stumble upon your own helpful ways of asking questions that work for you. It’s all part of the journey of prompt engineering.

Be Specific

Crafting the right prompts can be like finding the sweet spot in a good conversation, and it’s maybe the most crucial step to hitting it off with these text-generating systems. The more details, the better. You also need to be clear. Otherwise, the LLM may make assumptions or even hallucinate.

First, let’s take a look at some prompts that are too vague.

Prompt: Develop a feature to enhance data security.

Prompt: Can you build a tool to automate the process?

Prompt: Optimize the code.

Prompt: We need a function to process transactions.

The following are much more detailed and should get better results:

Prompt: Develop a Python function to parse dates from strings. The function should be able to handle the formats YYYY-MM-DD, MM/DD/YYYY, and Month DD, YYYY. It should return a datetime object. Provide a script that demonstrates the function handling at least three examples of each format correctly, along with a document explaining any dependencies, the logic used in the function, and instructions on how to run the script.

Prompt: Develop a SQL query to retrieve from our database a list of customers who made purchases above \$500 in the last quarter of 2023. The query should return the customer's full name, their email address, the total amount spent, and the date of their last purchase. The results should be sorted by the total amount spent in descending order. Please ensure that the query is optimized for performance.

Acronyms and Technical Terms

It's crucial to be clear with technical terms and acronyms while drafting a prompt. This jargon often means different things in different contexts and can lead to unhelpful responses. Thus, it's a good idea to spell out acronyms and give clear definitions or explanations of any technical terms used.

For example, suppose you are using ChatGPT to help resolve a database connection issue. A poorly crafted prompt might be:

Prompt: Having DB connection issues. How to fix it?

In this prompt, "DB" is ambiguous as it might refer to different database systems like MySQL, PostgreSQL, or others, and the nature of the connection issue is not clarified.

A more effective prompt would be:

Prompt: I am encountering a connection timeout issue while trying to connect to my PostgreSQL database using JDBC. How can I resolve this?

This prompt clearly spells out the database system in use, the method of connection, and the specific issue encountered.

NOTE

Mark Twain once [wrote](#), “The difference between the almost right word and the right word is really a large matter. ’Tis the difference between the lightning bug and the lightning.” In a way, the same thing can be said about writing a prompt.

Zero- and Few-Shot Learning

With *zero-shot learning*, you provide one prompt and get the answer you want. Often, this works fine. But given the complexities of programming languages and frameworks, there are times when you need to nudge the LLM.

You can do this with *few-shot learning*. This refers to an LLM’s capability to understand and perform a task with very few examples or training data. This is a significant advantage over traditional machine learning models, which may require a large amount of training data to perform adequately on a task. The LLM’s capability is primarily due to the extensive pretraining on a diverse range of internet text that the LLM undergoes before it is fine-tuned for a specific task.

Let’s take a look at an example of few-shot learning. Consider a scenario in which we want to generate a function that normalizes a given list of numbers. It will scale the values in the list to a range of [0, 1]. In the instructions, we include a list of a few examples of the inputs and normalized outputs.

Prompt: Based on the following examples of normalizing a list of numbers to a range of [0, 1]:

1. Input: [2, 4, 6, 8] Output: [0, 0.3333, 0.6667, 1]

2. Input: [5, 10, 15] Output: [0, 0.5, 1]

3. Input: [1, 3, 2] Output: [0, 1, 0.5]

Generate a function in Python that takes a list of numbers as input and returns a list of normalized numbers.

ChatGPT will “learn” from the data and come up with some code:

ChatGPT:

```
def normalize(numbers):
    min_value = min(numbers)
    max_value = max(numbers)
    if min_value == max_value:
        return [0] * len(numbers) # Handle the case where
                                   # all numbers are the same
    else:
        return [(x - min_value) / (max_value - min_value) for x in numbers]
# Test the function with one of the examples
print(normalize([2, 4, 6, 8])) # Output: [0.0, 0.3333333333333333,
                                   # 0.6666666666666666, 1.0]
```

Leading Words

The concept of *leading words* refers to specific keywords or phrases that can guide an LLM toward creating a particular kind of output. Sometimes you can achieve the desired result using just one code word. Here's an example:

Prompt:

Create a simple Python function that

1. Prompts me for a temperature in Fahrenheit

2. Converts Fahrenheit to Celsius

def

Using the word *def* as a leading word informs the model that it should begin writing a Python function. [Table 3-4](#) gives more examples of leading words.

Table 3-4. Examples of leading-word prompts

Context	Leading word
JavaScript function	Function
HTML element	<button
CSS styling	P {
SQL insert query	INSERT INTO
Java method creation	public

Chain of Thought (CoT) Prompting

In 2022, some Google researchers introduced *chain-of-thought (CoT) prompting* in their paper [“Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”](#). This approach enhances the reasoning abilities of LLMs by breaking down a complex problem into different steps. It’s actually similar to few-shot learning, which allows for nudging the model.

CoT prompting can be very useful in software code generation tasks. Let’s look at an example. Suppose you want to create a web application with a user registration and login functionality using Flask, a Python web framework. [Table 3-5](#) shows the CoT prompting steps.

Table 3-5. Chain-of-thought prompt examples

Action description	Prompt
Understand the requirement	“I need to create a web application using Flask. The application should have a user registration and login functionality. Where should I start?”
Set up Flask application	“Let’s begin by setting up a basic Flask application. How can I do that?”
Create user model	“Now that the Flask application is set up, I need to create a user model for handling registration and login. How should I structure this model?”
Implement registration	“With the user model in place, how can I implement a registration page with the necessary fields?”
Implement login	“Now let’s move on to creating a login page. How can I ensure secure login?”
Session management	“After a user logs in, how should I manage user sessions to keep users logged in as they navigate through the app?”
Logout implementation	“Finally, how can I implement a logout function to securely log users out of the application?”

Leading Questions

Leading questions in a prompt can often fetch wonky responses from the LLM. It’s better to stay neutral and unbiased. Also, it’s good practice to avoid making assumptions; spell things out instead.

This prompt is a leading question:

Prompt: Isn’t it true that migrating to a microservices architecture will always improve system scalability?

A more balanced prompt would be:

Prompt: What are the advantages and potential challenges of migrating to a microservices architecture in terms of system scalability?

Ask for Examples and Analogies

Suppose you don't know the concept of inheritance in object-oriented programming. You go to ChatGPT and enter this prompt:

Prompt: Explain inheritance that is used in object-oriented programming.

You will get a detailed response. But you may want to get something that's easier to understand. A good way to do this is by asking the LLM for an analogy:

Prompt: Explain inheritance that is used in object-oriented programming by using an analogy.

ChatGPT: Think of inheritance like a family tree, where children inherit certain traits and properties from their parents and, potentially, grandparents.

From there, ChatGPT provides more detail, which proceeds from the analogy, to explain the key elements of inheritance.

Reducing Hallucinations

In [Chapter 2](#), we learned that prompting an LLM can lead to a response that is a *hallucination*, such that the content generated is false or misleading but the LLM expresses the response as if it were true. Hallucinations can be particularly challenging for software development, which requires accuracy.

No doubt, applying the lessons in this chapter can mitigate this issue, but even a well-crafted prompt can spin up hallucinations. There are numerous reasons for this:

Lack of ground truth verification

LLMs generate responses based on patterns learned from training data without the ability to verify the accuracy or reality of the information.

Overfitting and memorization

LLMs might memorize incorrect or misleading information in their training datasets, especially if such data is repetitive or common.

Bias in training data

If the training data contains biases, inaccuracies, or falsehoods, the model will likely replicate these in its outputs.

Extrapolation and speculation

Sometimes, LLMs might extrapolate from the patterns they've seen in the data to generate information about topics or questions that were not adequately covered in the training data.

Lack of context or misinterpretation

LLMs can misinterpret or lack the necessary context to accurately respond to certain prompts. They may not fully understand the nuances or implications of certain queries.

Slang and idioms

Such language can create ambiguity that may lead the model to misinterpret the intended meaning, especially if it hasn't seen enough examples of the slang or idiom in context during training.

Then how to reduce hallucinations? For one thing, it's important to avoid asking open-ended questions like this:

Prompt: What are the different ways to optimize a database?

This type of prompt encourages the LLM to resort to speculation or over-generalization. The model may also misinterpret the intent of the question or the desired format of the answer, leading to responses that veer off-topic or contain fabricated information. There may actually be a cascade of hallucinations.

One effective technique is to provide a set of predefined options and ask the AI to choose from them. For example, the preceding prompt could be rephrased as follows:

Prompt: Which of the following is a method to optimize a database: indexing, defragmenting, or compressing?

As another example, consider asking the LLM for a certain type of conclusion. Here is an effective prompt:

Prompt: Is the following syntax correct for initializing an array in Java? Provide a “yes” or “no” response.

Or you can include multiple steps in the prompt to better guide the model through a structured process and narrow down the possibilities for straying off course:

Prompt:

Step 1: Create a Fibonacci sequence generator.

Step 2: Use the iterative method.

Step 3: Write a Python function named `generate_fibonacci` that takes an integer `n` as an argument.

Step 4: The function returns the first `n` numbers in the Fibonacci sequence as a list.

Security and Privacy

Being watchful about security and privacy while crafting prompts is key. In fact, the duty to take appropriate precautions should be in the company rulebook. It's crucial to steer clear of any sensitive or personal information, such as personally identifiable information (PII) in your prompts. Here's an example of a prompt that contains identifying information:

Prompt: How would you fix a login issue reported by John Doe at john.doe@example.com?

It's wiser to go with something like:

Prompt: How would you tackle a login issue reported by a user?

This keeps private information private.

It's also smart to steer clear of spilling any sensitive system details in the prompts. Avoid this:

Prompt: How to fix a database connection error on our production server at IP 192.168.1.1?

Instead, it's safer to use a more generic question:

Prompt: How to fix a generic database connection error?

Moreover, make sure your prompts don't accidentally nudge folks toward shady practices. A prompt like this is fine from a security viewpoint:

Prompt: How to detect and prevent SQL injection?

But not this one, which might stir up some bad intentions:

Prompt: How to exploit SQL vulnerabilities in a website?

Besides sticking to security and privacy rules, embracing diversity and inclusion when making prompts is important. Getting a solid grasp on bias, which often reflects the training data, is key. It's a good call to use neutral and inclusive language to avoid any discriminatory or exclusionary phrases in the prompts. Also, getting feedback from a diverse group of people on your prompt crafting can help. This not only improves fairness and inclusivity when interacting with the LLM but also helps get a more accurate and well-rounded understanding of the topics at hand.

Autonomous AI Agents

We've seen how you can nudge LLMs to map out the steps for a process. That's at the heart of code generation.

But AI agents can crank it up a notch. They don't just follow prompts. They get creative with LLMs to figure out a game plan for whatever goal you toss at them, and they tap into specialized databases like Pinecone and Chroma DB. They handle complex word embeddings, which the models understand.

Autonomous AI agents are based on academic research and are usually part of open source projects. Their real power is automation. To see how this works, let's take an example. Suppose you set the objective as follows:

Prompt: Create a basic weather application with a user login system.

Table 3-6 shows a process that an autonomous agent may go through.

Table 3-6. Process for an autonomous agent

Phase	Tasks
Creation tasks	Design the user interface (UI). Sketch the basic layout of the dashboard. Select color schemes and fonts. Design icons and other graphic elements.
API integration for weather data	Search the internet for reliable weather data APIs. Determine the data points to be displayed. Write code to fetch and update weather data.
Location selection functionality	Create a search bar or dropdown for users to select their location. Connect this to the API code.
Error handling	Handle errors like failed API calls or invalid location entries.
Prioritizing tasks	Prioritize setting up the API integration. Focus on the UI. Work on location selection functionality and error handling.
Iteration	Review the generated code and the current state of the weather dashboard. Identify any remaining tasks or new tasks that have arisen during execution. Repeat the create and prioritize steps.

This technology is at the forefront and holds much promise. However, it’s not without its fair share of hurdles:

Being resource hogs

Agents can guzzle down large amounts of compute power. This can put the squeeze on your processors and databases, leading to more wait time, less reliability, and a slump in how things run as time goes on.

Getting stuck in infinite loops

Sometimes agents just run in circles, thanks to a lack of progression or a repetitive reward system.

Being experimental

Agents can be rough around the edges. They might come with a few bugs or unexpected behaviors and might not be quite ready for the big leagues, depending on what you need them for.

Having amnesia

Agents may simply forget certain steps or instructions.

Having difficulty handling a large number of tasks

Got a whole laundry list of tasks? That might trip up these agents.

Getting distracted by extraneous details

Agents might get sidetracked by the little things that don't matter, which could send them down the wrong path when picking tools to use.

Another innovation that bolsters LLMs is retrieval augmented generation (RAG). With RAG, a generative AI application—say written in a framework like LangChain—accesses external sources of data, usually vector databases. They provide more grounding of the model in specific knowledge, which should enhance the LLM's responses.

RAG can be particularly useful when handling complex software development tasks, such as in these scenarios:

Tackling bugs and glitches

When developers encounter bugs or errors, RAG digs up fixes and workarounds from all around the web, looking in places like forums or bug databases. It can whip up some tailor-made solutions or code patches that fit your problem.

Spicing up code reviews

RAG can pull in all the coding best practices, standards, and must-follow rules from a company's internal resources. This means it can help streamline your code reviews, dishing out tips and tricks to make your code shine.

When it's time to put your code through its paces, RAG can be your pit crew. It finds all sorts of test scenarios and patterns, tweaks them to suit your project's needs, and helps you roll out test cases or scripts faster.

NOTE

In some cases, an LLM will indicate that there's an issue with your prompt. For example, suppose you write this:

Prompt: Write a REST API using assembler.

This is how ChatGPT responded:

ChatGPT: Creating a REST API using an assembler language is highly unconventional and complex due to the low-level nature of assembler languages. These languages are usually used for tasks that require fine-grained control over hardware, or for scenarios where performance is a critical concern.

Conclusion

Again, crafting the perfect prompt involves mixing science with a splash of creativity. It's all about finding the right ingredients—some creativity, a bit of gut feeling, and a structured approach—to cook up prompts that get LLMs to serve up what you want. No magic recipe exists, but if you're clear, throw in a few examples, and lay out your prompts well, you're on track for better answers.

It's a process, really. You try something, see how it goes, tweak it, and try again. And as with any skill, you get better the more you work on it with different topics and tasks.