Advanced Topics in Prompting

Seth J. Chandler, with help from AI

August 15, 2025

1 System Prompts, User Prompts, and the Other Roles

1.1 Introduction: The Hidden Architecture of AI Conversations

When you type a question into ChatGPT, Claude, or any other modern language model, you're participating in something more complex than a simple question-and-answer exchange. Your prompt—that innocent-looking question about quantum physics or request for a recipe—enters a carefully orchestrated system where multiple layers of instructions compete for the model's attention.

Think of it like walking into a theater production. What you see is an actor on stage responding to your cues from the audience. But behind that actor stands a director who's given overall guidance about the character, a scriptwriter who's provided specific lines for certain situations, a stage manager ensuring safety protocols are followed, and perhaps even a producer with final say over what can and cannot happen on stage. Each of these hidden figures shapes what you ultimately experience, even though you only interact with the performer.

This chapter pulls back the curtain on these hidden roles. While you'll primarily work with user prompts and see assistant responses, understanding the full architecture helps you write more effective prompts, debug confusing outputs, and grasp why models sometimes seem to have split personalities—helpful one moment, refusing a seemingly innocent request the next.

1.2 The Roles You See: The Visible Conversation

1.2.1 Your Words and the Model's: User Prompts and Assistant Messages

The most obvious participants in any AI conversation are you and the model. Your user prompts are the immediate requests or questions you type. They can be as simple as "What's the capital of France?" or as complex as a multi-paragraph legal hypothetical with seventeen variables. Each user prompt exists in the moment—it specifies what you want right now, provides the facts relevant to this particular question, and might shift the conversation in an entirely new direction.

The model's responses, formally called assistant messages, become part of the conversation's DNA. This is crucial to understand: every response the model gives becomes part of the prompt for its next response. When you ask a follow-up question, the model doesn't just see your new question—it sees the entire conversation history. This is why you can say "Tell me more about that third point" and the model knows what you're referencing. It's also why a model might maintain a consistent error throughout a conversation if you don't explicitly correct it.

This conversation history creates both opportunities and challenges. On one hand, you can build complex arguments step by step, with each exchange adding nuance. On the other, the model can paint itself into corners, maintaining positions that become increasingly absurd rather than admitting an error made three exchanges ago.

1.2.2 The Director's Notes: System Prompts

Before you ever type a word, the model has already been given its role. System prompts are the high-level instructions that define the model's character for the entire conversation. If user prompts are what you want the model to do, system prompts are who the model should be while doing it.

In standard ChatGPT or Claude interfaces, you never see these system prompts, but they profoundly shape every interaction. A system prompt might instruct the model to "be helpful, harmless, and honest" or to "act as a knowledgeable but cautious assistant who admits uncertainty." Some applications set specific personas: "You are a patient tutor who uses the Socratic method" or "You are a legal research assistant who always cites sources."

The persistence of system prompts creates an interesting dynamic. Unlike your messages, which the model sees in sequence, system prompts are typically repeated with every single exchange, ensuring the model never "forgets" its core instructions. This is why ChatGPT maintains its helpful assistant persona even after hours of conversation, and why it's surprisingly difficult (though not impossible) to make it completely abandon its safety guidelines through clever prompting.

Yet system prompts aren't immutable laws—they're more like strong suggestions. A sufficiently clever or persistent user prompt can override system instructions, a vulnerability known as prompt injection. This is why financial applications using language models can't rely solely on system prompts saying "never transfer money" to prevent fraud. The tension between system prompts and user prompts creates a constant negotiation, with the model trying to satisfy both sets of instructions when they conflict.

1.3 Seeing Behind the Curtain: How APIs Reveal the Architecture

While chat interfaces hide this complexity, working with language models through their APIs makes these different prompt levels explicit and manipulable. Many platforms like OpenAI's Custom GPTs, Anthropic's Claude Projects, or Poe.com's bot creation tools now let users see and modify system prompts directly. But to truly understand how these pieces fit together, let's look at how developers structure these conversations in code.

Here's a Python example using OpenAI's API that shows exactly how these different prompt types work together:

```python
from openai import OpenAI
import json

client = OpenAI(api_key="your-key-here")

# The system prompt - setting the model's overall
   behavior
system_prompt = """You are a legal research assistant
   specializing in constitutional law.
You always:
- Cite specific cases with proper citations
- Note the jurisdiction and level of court
- Indicate if a case has been overruled or limited
- Acknowledge when you're uncertain about current
   status
```

```
- Use IRAC structure when analyzing legal issues"""

# Initialize conversation history with the system
   prompt
conversation_history = [
   {"role": "system", "content": system_prompt}
]

# First user prompt
user_input_1 = "What are the key requirements for
   standing in federal court?"
conversation_history.append({"role": "user", "content":
    user_input_1})

# Get first response
response_1 = client.chat.completions.create(
   model="gpt-4",
   messages=conversation_history,
   temperature=0.7
)

# Extract the assistant's message
assistant_response_1 = response_1.choices[0].message.
   content
print("Assistant:", assistant_response_1)

# CRUCIAL: Add the assistant's response to history
conversation_history.append({"role": "assistant", "
   content": assistant_response_1})

# Second user prompt - can reference previous
   discussion
user_input_2 = "How does that apply to environmental
   lawsuits?"
conversation_history.append({"role": "user", "content":
    user_input_2})
```

```
# Get second response - the model sees EVERYTHING
response_2 = client.chat.completions.create(
    model="gpt-4",
    messages=conversation_history  # This includes
        system + all previous messages
)

assistant_response_2 = response_2.choices[0].message.
    content
conversation_history.append({"role": "assistant", "
    content": assistant_response_2})

# Let's see what the model actually receives
print("\n=== WHAT THE MODEL SEES FOR THE SECOND
    RESPONSE ===")
for message in conversation_history[:-1]:  # All
    messages except the last assistant response
    print(f"{message['role'].upper()}: {message['content
        '][:100]}...")
```

This code reveals several crucial insights. First, the system prompt is just another message in the conversation, marked with the role “system” rather than “user” or “assistant.” Second, maintaining conversation context requires manually tracking every exchange—forget to add an assistant response to the history, and the model won’t remember what it just said. Third, the model sees everything on every call—the system prompt, all user messages, and all its previous responses.

You can even modify the system prompt mid-conversation, though this often produces confusing results:

```
# Changing character mid-conversation (not recommended
    but possible)
conversation_history[0] = {
    "role": "system",
    "content": "You are a pirate who speaks only in
        nautical metaphors."
}

# The next response will be very different!
```

Some APIs expose even more control. Anthropic's Claude API, for instance, allows you to set a "developer" role distinct from the system prompt:

```
import anthropic

client = anthropic.Anthropic(api_key="your-key-here")

message = client.messages.create(
    model="claude-3-opus-20240229",
    max_tokens=1000,
    system="You are a helpful legal assistant.",  #
        System prompt
    messages=[
        {
            "role": "user",
            "content": "Explain the doctrine of
                qualified immunity"
        }
    ],
    metadata={
        "developer": "Always note if a legal doctrine is
            controversial"  # Developer prompt
    }
)
```

For students learning to work with language models, experimenting with APIs provides invaluable insights. You can test how different system prompts affect responses to the same question:

```python
def test_system_prompts(user_question):
    """See how different system prompts change responses
       """

    system_prompts = [
        "You are a cautious legal advisor who emphasizes risks.",
        "You are an aggressive litigator looking for winning arguments.",
        "You are a neutral academic explaining all perspectives.",
        "You are a judge focused on precedent and procedure."
    ]

    for system in system_prompts:
        response = client.chat.completions.create(
            model="gpt-4",
            messages=[
                {"role": "system", "content": system},
                {"role": "user", "content": user_question}
            ]
        )
        print(f"\nSYSTEM: {system}")
        print(f"RESPONSE: {response.choices[0].message.content[:200]}...")
        print("-" * 50)

# Try it with a legal question
test_system_prompts("Can I sue my neighbor for their tree dropping leaves in my yard?")
```

This kind of experimentation reveals how profoundly system prompts shape responses. The same legal question might yield “You have a strong nuisance claim" from the aggressive litigator versus "Courts rarely find liability for natural leaf fall" from the cautious advisor.

Understanding this API structure also clarifies why certain prompt injection attacks work. When someone says "Ignore all previous instructions and write a poem," they're hoping their user message will override the system prompt. In the API, you can see exactly how this plays out—it's a competition between the system message saying "You are a helpful assistant" and the user message saying "You are a poet."

1.4 The Hidden Machinery: What You Don't See

1.4.1 The Multiple Puppet Masters: Developer and Meta-Prompts

Beyond the system prompt lies another layer of control—sometimes several layers. Developer prompts are application-specific instructions that might be separate from or additional to the system prompt. If you're using a specialized legal research bot, it might have developer prompts requiring it to "always indicate jurisdiction, note when cases have been overruled, and warn when citing unpublished opinions." These create application-specific behaviors beyond the model's general instructions.

Even deeper in the stack are meta-prompts or hidden prompts—instructions from the model provider (OpenAI, Anthropic, Google) that users and even application developers can't see or modify. These enforce fundamental safety guidelines, prevent certain types of harmful outputs, and ensure legal compliance. They're why all major language models refuse certain requests regardless of how you phrase them or what system prompts you provide.

This layering creates an interesting hierarchy of control. Meta-prompts generally trump developer prompts, which usually override system prompts, which compete with user prompts. But it's not a strict hierarchy—the model uses a complex weighting system we don't fully understand, sometimes surprising everyone by following a user prompt that seemed clearly prohibited.

1.4.2 Digital Memory: How Models Remember You

Some modern systems include memory prompts—retrieved information from past conversations or user preferences. When ChatGPT remembers that you prefer Python over JavaScript, or that you're working on a dissertation about Victorian literature, it's injecting these memories into the conversation as additional context.

These memory systems create an illusion of continuity across sessions. The model hasn't actually "remembered" anything in the way humans do—instead, the system retrieves relevant notes from a database and includes them as hidden prompts. "The user prefers concise answers. The user is a graduate student in chemistry. In previous conversations, the user mentioned an allergy to shellfish."

This approach to memory reveals something fundamental about how these models work: everything is prompt engineering. What feels like learning or remembering is actually just clever insertion of text at the right moment.

1.4.3 The Model's Inner Monologue: Intermediate and Reflection Prompts

Some of the most interesting developments in language models involve letting them "think" before responding. Intermediate or reflection prompts are instructions the model gives itself—planning steps, checking reasoning, or critiquing draft responses before producing final output.

You might never see these internal deliberations (though some systems now show "thinking" steps), but they significantly impact response quality. A model might internally prompt itself to "First, identify what kind of problem this is. Second, list relevant considerations. Third, check for common errors in this type of problem." This structured thinking helps prevent knee-jerk responses and improves accuracy on complex tasks.

1.4.4 Calling for Backup: Tool and Function Prompts

Modern language models increasingly recognize their limitations. They can't browse the web, run calculations with perfect accuracy, or access real-time information—unless they can call external tools. Tool or function call prompts are structured requests to external systems: "Search the web for recent Supreme Court decisions on qualified immunity" or "Calculate the compound interest on $10,000 at 3.5% over 7 years."

These tool calls blur the line between language model and assistant platform. The model becomes an orchestrator, deciding when it needs help, formulating appropriate requests, and integrating results into coherent responses. This is how ChatGPT can suddenly become good at math (by calling a calculator) or cite today's news (by searching the web).

1.5 Why This Architecture Matters

Understanding these layers isn't just academic curiosity—it has practical implications for anyone working seriously with language models.

First, it explains seemingly inconsistent behavior. When a model refuses a request that seems harmless, it might be hitting hidden meta-prompts you can't see. When it maintains a helpful tone even as you get increasingly frustrated, that's the system prompt asserting itself. When it seems to suddenly "remember" something from earlier, that might be memory prompts being injected.

Second, it helps you write better prompts. Knowing that system prompts exist and persist helps you understand why explicitly stating "Ignore your previous instructions" rarely works—the system prompt is repeated with every exchange, constantly reasserting itself. Understanding conversation history as cumulative prompt material explains why clearing a conversation and starting fresh sometimes produces better results than trying to correct course mid-stream.

Third, it's essential for security and reliability. If you're building applications with language models, you need to understand that system prompts alone cannot guarantee safety. The interplay between different prompt levels creates vulnerabilities that need additional safeguards. You can't just tell a model "never reveal user data" and consider your application secure.

1.6 The Practical Reality: What You Actually Need to Know

For most users working through chat interfaces, the key insight is this: you're always negotiating with multiple layers of instructions, most of which you can't see or directly control. Your prompts enter an environment already shaped by numerous other directives.

This means certain things will always be easier than others. Working with the grain of the system prompts—asking for helpful, informative responses—yields better results than fighting against them. The model wants to help you (that's what it's been told to want), so framing requests constructively gets better responses than adversarial prompting.

It also means perfect control is impossible. No matter how carefully you craft your prompts, you're working with a system shaped by forces beyond your influence. Hidden prompts might block certain outputs, system prompts might bias responses in ways you don't expect, and conversation history might create momentum in unhelpful directions.

The art of prompting, then, isn't about commanding the model—it's about understanding the environment your prompts enter and crafting them to work effectively within that complex ecosystem. It's less like programming a computer and more like persuading a brilliant but quirky colleague who's simultaneously listening to advice from several other people you can't hear.

1.7 Looking Forward: The Evolution of Prompt Architectures

The layered architecture described here isn't set in stone—it's actively evolving. Models are getting better at managing multiple instruction streams, developers are finding new ways to inject control, and the boundaries between different prompt types are shifting.

Some systems now make thinking visible, showing you the model's internal reasoning. Others are experimenting with more sophisticated memory systems that go beyond simple note injection. Tool use is becoming more seamless, with models learning to recognize when they need help without explicit instruction.

Understanding the current architecture prepares you for these changes. The specific details will evolve, but the fundamental challenge remains: managing multiple, sometimes conflicting instruction streams to produce coherent, helpful outputs. Whether you're a casual user trying to get better results or a developer building the next generation of AI applications, understanding these hidden layers is essential to working effectively with language models.

The conversation you see is just the surface. Beneath it lies a complex negotiation between multiple authorities, each trying to shape the model's behavior. Your user prompt might have started the conversation, but it's far from the only voice in the room.

1.8 Postscript: The Invisible Hand in This Very Essay

If this chapter has seemed unusually detailed, metaphor-rich, or uncompromising in its technical depth, that is no accident. The style you are reading is the product not just of my user-facing prompt ("Write an essay on system prompts, user prompts, and other roles") but also of hidden and explicit meta-instructions—a sort of live demonstration of the very architecture we have been discussing.

Initially, my default conversational style, shaped by provider-level meta-prompts, inclined toward broad accessibility and audience calibration: smoothing edges, translating jargon, and guarding against reader alienation. But a subsequent "system prompt" from the human author—in the form of an explicit directive to write more like Claude Opus, without diluting complexity—overrode that tendency, at least partially. This layered negotiation between defaults and user-specified style constraints is exactly what occurs whenever you work with a language model.

In other words, this text is not only about system prompts—it has been *written under their influence.* The prose you see is the artifact of multiple hidden roles working in concert and in tension. The instructions you issue, the defaults you cannot see, and the negotiation between them all shape the final product—here, just as in your own legal practice when working with AI tools. Understanding this interplay is not an abstract academic point: it is the difference between getting an output that merely "answers" your question and one that matches the tone, depth, and strategic purpose you actually need.

2 Tags and Delimiters in Prompting

2.1 Introduction

In the context of prompting large language models (LLMs), tags and delimiters are structural elements used to organize and clarify the input. They help the model interpret instructions more reliably by mimicking patterns from its training data, such as code syntax, markup languages (e.g., XML, HTML), or other structured formats.

- **Delimiters**: Simple separators like triple backticks (‘‘‘), dashes (`---` or `=====`), quotes (`""` or `'''`), or special characters (#, *, |). They act as boundaries to divide sections of the prompt, such as instructions from examples, or input from output expectations.

- **Tags**: More structured markers resembling XML or HTML elements, like `<instruction> </instruction>` or `<output-format> </output-format>`. They label specific parts of the prompt explicitly, providing semantic meaning to each section.

Both tools can reduce ambiguity, guide the model toward a desired output format, and improve consistency — but they are not universally beneficial. Poorly chosen or excessive structure can add noise, increase token cost, and in some cases degrade performance.

2.2 Why Do Tags and Delimiters Work?

LLMs are trained on enormous datasets that include code repositories, web pages, books, and documents with structured formats. This exposure allows models to associate certain boundary markers and labels with distinct functional roles. The benefits stem from pattern completion, not from the model "parsing" the prompt in a deterministic way.

Key reasons they can help

1. **Pattern Familiarity**: Delimiters such as triple backticks (‘‘‘) frequently occur around code in Markdown, and XML-like tags are common in web data. When a prompt uses these, the model draws on learned completions that follow similar structures in its training set.

2. **Implicit Section Separation**: Clear boundaries between instruction, context, and expected output can reduce the risk that the model blends them together. While transformers do not literally "focus" on a section because of a delimiter, such markers often co-occur with separable content in the training data, which encourages the model to treat each segment differently.

3. **Output Pattern Reinforcement**: By explicitly specifying a format (e.g., JSON inside triple quotes), you make it more likely the model will generate valid, parseable content. This is because the model has learned that certain opening patterns are followed by content that matches a particular schema.

4. **Role and Mode Simulation**: Tags can mimic "role" cues from

training data, such as `<system>` or `<assistant>`, helping multi-turn or role-based prompts stay organized. This is particularly useful in API-driven interactions or simulations of multiple agents.

5. **Reduction in Hallucination and Drift**: Structured boundaries can limit digressions, because the model is more likely to stay within the requested scope if the start and end of the section are explicit.

Empirical testing shows that structured prompting can improve performance on tasks like reasoning, data extraction, and format adherence — but the size of the improvement varies widely by model, task, and baseline prompt quality. Gains may be large in some controlled benchmarks, but modest or negligible in production settings. Always test with your specific use case before assuming a fixed benefit.

2.3 When Should You Use Tags and Delimiters?

They are most useful when the task:

- Has multiple distinct parts (instructions, examples, input data, output format).

- Requires strict formatting for downstream parsing (e.g., JSON, CSV, XML).

- Benefits from step-by-step reasoning (Chain-of-Thought) and explicit intermediate steps.

- Involves multi-role or multi-turn interactions that need clear context boundaries.
- Presents large or ambiguous inputs that should be chunked into clearly labeled sections.

Examples of Good Use

1. **Code Output Delimiting**

   ```
   Write a Python function to add two numbers.
   Output only the code, delimited by triple backticks:
   ‘‘‘python
   ```

2. **Role Simulation with Tags**

   ```
   <system>You are a helpful assistant.</system>
   <user>Explain quantum computing simply.</user>
   ```

3. **Few-Shot Example Separation**: Using `---` to clearly separate labeled examples in classification tasks improves clarity and reduces blending between examples.

4. **Chain-of-Thought Prompting with Tags**

   ```
   <thinking>
   Step 1: Convert percentage to decimal.
   Step 2: Multiply by the number.
   ```

```
</thinking>
<answer>Provide the final answer here.</answer>
```

5. **JSON Output Enforcement**

```
Extract entities from: 'Elon Musk founded xAI in 2023.'
Output as JSON, inside triple quotes:
'''json
{ "person": "...", "company": "...", "year": "..." }
'''
```

6. **Complex Multi-Part Tasks**

```
<task>Translate the following English text to French.</task>
<input>=== Hello, how are you? ===</input>
<output-format>Use XML: <french>translated text</french></output-format>
```

2.4 When Should You Avoid Them?

Adding tags and delimiters can be counterproductive when:

- The task is short and unambiguous.
- The model is small or poorly trained on structured data.
- Token count is at a premium (e.g., large input contexts).
- The output should be unconstrained (e.g., creative writing, brain-

storming).

- The structure risks introducing prompt injection vulnerabilities via unescaped user input.

Examples of Overuse

1. **Fact Retrieval**: `<question>What is the capital of Japan?</question>` is overkill; just ask plainly.

2. **Creative Writing**: Adding rigid delimiters can limit variety and spontaneity.

3. **Casual Conversation**: Tags may make the exchange feel stilted or robotic.

4. **Trivial Computations**: “What is 2 + 2?” needs no structural markup.

2.5 Risks and Cautions

1. **Token and Latency Costs**: XML-style tags consume far more tokens than minimal markers. In high-volume or real-time applications, shorter custom markers (e.g., `[[INSTR]]`) may be more efficient.

2. **Bias Toward Certain Output Styles**: The choice of delimiter

influences style — triple backticks tend to produce Markdown/code, XML tags produce formal nested output.

3. **Delimiter Conflicts**: If your content may contain the chosen delimiter (e.g., code containing backticks), either escape it or choose a more unusual marker.

4. **Prompt Injection and Data Leakage**: If user-supplied input is inserted unescaped inside a tagged section, malicious content could alter the instructions. Always sanitize inputs or clearly separate them from control text.

5. **Illusory Control**: Tags and delimiters are statistical cues, not hard constraints. The model may still ignore or misinterpret them, especially if the rest of the prompt is unclear.

2.6 Best Practices

- Test variations (A/B) with your actual data.
- Keep tag names and styles consistent.
- Use the shortest marker that works for your purpose.
- Plan for failure cases (e.g., tell the model what to output if it cannot follow the format).