A Generalised Guide to Prompt Engineering

Principles

An agnostic guide to prompting, drawing on Anthropic's prompt engineering tutorial and the Learn Prompting guide.

Author: Antonio Lobo-Santos

WATERMARK: This document was created by Antonio Lobo-Santos - AI Coding Seminar

---

Table of Contents

- A Generalised Guide to Prompt Engineering
  - Principles
  - Table of Contents
  - Prompt Hierarchy
    - System vs. User Prompts
  - Foundational Principles: Clarity and Intent
    - The Anatomy of a Prompt
    - Being Clear, Direct, and Positive
    - The Power of Role Personas
  - Intermediate Techniques: Structure and Reasoning
    - Separating Instructions from Data
    - Guiding Output Format and Style
    - Chain-of-Thought Reasoning
    - Few-Shot Prompting with Examples
  - Advanced Applications and Safeguards
    - Mitigating Hallucinations
    - Building Complex Prompts
  - Beyond a Single Prompt: Advanced Workflows
    - Prompt Chaining
    - Augmenting LLMs with External Tools
    - Search and Retrieval (RAG)
  - Resources:
  - References:

Prompt Hierarchy

System vs. User Prompts

Modern LLMs use a prompt hierarchy to structure conversations. Typically, a system prompt (or system message) sets the global context, constraints, and style for the model, while the user prompt contains the

specific task or question within that context. The system prompt essentially acts as a contract or rules-of-engagement that the assistant must follow throughout the conversation.

- **System Prompt:** Defines the AI's identity, role, and high-level directives. This is where you state the model's persona (e.g. "You are an expert technical writing assistant"), overall tone, and forbidden behaviors. Once set, these guidelines implicitly apply to all subsequent responses. The system prompt should establish boundaries (what the AI should or shouldn't do) and any persistent style or ethical guidelines. It preloads context before the user's actual query is considered.
- **User Prompt:** Contains the actual user query or task, often with additional context or data. The user prompt should be clear and explicit about what is needed, leveraging the conditions set by the system prompt. Generally, all task-specific instructions and data belong in the user prompt, not the system prompt. This separation ensures the model first adopts the right role/policies (system level), then focuses on executing the requested task under those conditions.

Example:

> **System:** "You are a seasoned data science assistant. Respond in a concise, formal tone and refuse to speculate beyond provided information." **User:** "Analyze the following dataset for anomalies and summarize your findings in bullet points."

(Here the system prompt sets an expert persona and constraints, while the user prompt provides the task and data to analyze.)

The synergy between system and user prompts ensures the model stays in character and on task. The system prompt keeps the model focused and consistent in style, and the user prompt drives the specific content. Used correctly, this hierarchy leads to more coherent and reliable outputs.

Foundational Principles: Clarity and Intent

The Anatomy of a Prompt

A well-crafted prompt is composed of distinct components, each serving a role in guiding the model's output. Not every prompt needs every component, but understanding them helps in structuring effective prompts:

1. **Directive** - The core instruction specifying the task. This is what you want the model to do or answer. A clear directive is crucial; without it, the AI may respond irrelevantly.
2. **Context / Additional Information** - Background details or data needed to perform the task. This could be a text passage to summarize, a user query history, or facts the model should use. Including relevant context ensures the AI has the information required to produce a correct answer
3. **Role / Persona** - An optional identity for the model to adopt (e.g. teacher, translator, Python interpreter). Assigning a role helps the model shape its tone and domain knowledge appropriately
4. **Examples (Few-Shots)** - Demonstrations of desired behavior. By showing input output examples, you implicitly teach the model the pattern or format you expect.
5. **Output Formatting** - Explicit instructions on how the answer should be structured or formatted (bullet points, JSON, code block, etc.). This guides the model's presentation of the answer

These elements can be assembled in different orders. A commonly effective pattern is to provide any examples and context first, and place the main directive last. Ensuring the primary instruction comes at the

end of the prompt helps the model focus on the actual task after processing all prior details. In other words, the model reads all the context and examples, then finally sees "Now, do X" - reducing the chance it continues rattling off context instead of executing the task.

Prompt Example (Well-Structured):

```
**Role:** You are a science communicator.
**Context:** Audience is high-school students with basic biology
knowledge.
**Examples:** (Q&A style example provided)_
**Formatting:** Answer in exactly 3 bullet points, simple language.
**Directive:** Explain how DNA stores genetic information.
```

Why it works: The role ensures an appropriate tone, the context clarifies who the answer is for, an example (not shown above for brevity) would illustrate the expected level and style, the formatting requirement fixes the output structure, and the directive at the end makes the task explicit only after all context is established.

Bad Example (Poorly Structured):

> Explain DNA to a student. It's the molecule with genetic info. Give 3 points. You are a science teacher.

What's wrong: The instructions are jumbled and the directive ("Explain...") appears at the start, before context ("genetic info") and role ("you are a teacher") are fully given. The model might start answering too early or ignore some instructions. There's also minimal guidance on style or depth.

In summary, think of a prompt as a mini-program for the AI: declare any required setup (role), feed in data (context, examples), specify output requirements, and finally call the main instruction. This modular approach yields far more predictable results than a single run-on sentence.

Being Clear, Direct, and Positive

LLMs are literal and do exactly what you ask - so phrasing matters. Ambiguous or under-specified prompts (e.g. "Make this better") often produce unsatisfactory results. Effective prompts use precise language and leave little room for interpretation. Some best practices:

- **Be Specific:** State exactly what output you want. Instead of "Tell me about climate change", say "In 100 words, summarize the impact of climate change on coastal cities, focusing on rising sea levels." (specifying length, topic focus, and angle). Specific directives lead to focused, relevant answers.
- **Use Strong Action Verbs:** e.g. "List three causes of X", "Compare A and B", "Convert this JSON to YAML". Avoid vague verbs like "handle" or "discuss" without clarity.
- **Give Explicit Constraints:** If you have requirements on length, format, or style, mention them. For example, "Respond in Markdown table format," or "Limit the answer to one paragraph." If you need only code as output, say so clearly
- **Prefer Positive Phrasing:** It's more effective to tell the model what to do rather than what not to do. For instance, use "Write in an active voice." instead of "Do not use passive voice." Models can sometimes ignore negations ("do not") but will follow affirmative instructions for desired style or content. Guiding the behavior works better than only forbidding certain things.

- **Include Success Criteria or Checks:** If applicable, describe what a good answer should achieve (e.g. "The solution should include at least one real-world example and cite a source."). This helps the model self-evaluate its response against the criteria.

Bad Example (Unclear): "Explain this better." **What's wrong:** Too vague what to explain? Better in what sense (shorter? more formally? more detail?) The AI may guess incorrectly what you want improved.

Improved Prompt: "Rewrite the following explanation in clearer, simpler terms for a general audience, in under 3 sentences." **Why it's better:** Specifies the transformation (simplify the explanation), the target audience (general public), and limits scope (3 sentences). The model isn't left to guess your intent.

In short: clarity is king. A good mental check is Anthropic's "new colleague test" if a new employee with no context would be confused by your instructions, the AI likely will be too. Take the time to spell out exactly what you mean in straightforward language.

The Power of Role Personas

One of the most powerful prompt tools is assigning the AI a persona or role to play. By front-loading a role, you constrain the style, tone, and knowledge domain of the model's responses. This is often called "role prompting" or "persona injection." For example: "You are a senior tax law attorney...", or "Act as a customer support chatbot for a banking app...".

A strong persona description typically includes:

- **Role & Goal:** What the AI is and what it should optimize for. (E.g. "You are a helpful coding assistant who explains solutions clearly to junior developers.")
- **Expertise or Domain Knowledge:** Frame the AI's knowledge background (real or fictional). This can dramatically improve factual and contextual accuracy when answering domain-specific questions. For instance, a "medical expert" will respond with more technical detail on health topics.
- **Tone & Style Guidelines:** Specify formality, dialect, or other stylistic preferences. A role can imply style (a "friendly librarian" vs "military drill instructor" will naturally adopt different tones).
- **Constraints/Biases:** If needed, mention any limits (e.g. "As a medical advisor, do not provide diagnosis, only general info.").
- **Example Behavior:** Optionally, give a short example of how this persona might respond to something, illustrating the desired style.

Role prompting can significantly boost accuracy and relevance on complex tasks by focusing the model's attention. It also helps keep the model "in character" for multi-turn conversations, making the experience more consistent for end-users.

Example Persona Prompt:

> **System:** "You are CodeGuru, an AI programming assistant. You have expertise in Python and software engineering best practices. You provide step-by-step explanations and well- commented code solutions. You never just give the final code you first explain your approach." **User:** "How do I sort a list of dictionaries in Python by a value?"

Result: The assistant responds as a Python expert teacher, perhaps explaining the sorted() function and giving a sample code snippet with comments.

Bad Persona Example: "You are a historian or maybe a scientist. Answer nicely." **Why it's bad:** It's ambiguous ("historian or scientist?") and provides no concrete guidance on style or depth. The model may ignore an ill-defined persona.

Caution: While personas are powerful, avoid overly stereotyped or biased roles. Be specific and factual in defining the role (focus on skills, not stereotypes). And remember that a role prompt doesn't grant the model actual credentials it only influences style and knowledge emphasis. Always combine role prompts with factual context or retrieval for domains requiring up-to-date information.

Intermediate Techniques: Structure and Reasoning

Separating Instructions from Data

When providing substantial input data along with your instructions, it's critical to clearly separate the two. This avoids any accidental mixing of what the model should use (data) vs. what it should do (instructions). The model should never confuse a user-provided text with an instruction from the user. Use delimiters or markup to isolate different parts of the prompt. Anthropic's guidance suggests employing XML-like tags for this purpose, but you can also use Markdown code fences or triple quotes.

For example:

```
 User: Summarise the text in the <article> below in one sentence.
 <article>
 [Full text of article goes here...]
</article>
```

By wrapping the article in a `<article>...</article>` tag (or similarly triple quotes), we make it obvious which text is source material and which is the actual task. The model is then less likely to treat the article's content as part of the instructions. This practice improves accuracy and reduces errors from misinterpreting prompt segments.

Using structured markup or Markdown for prompts often yields the best results. Many modern LLMs have been trained on data that includes markdown and HTML/XML patterns, so they respond well to well-formatted prompts. For instance, delimiting a code block or dataset with triple backticks or tags like `<data>...</data>` helps the model recognise it as an object to use or transform, not to be expanded or altered.

Example:

```
 Role: You are an API documentation assistant.
 Instructions: Read the JSON in <input> and explain each field
 briefly.
 <input>
 {
 "name": "Acme Rocket",
 "thrust": 300,
 "stages": 2
```

```
}
</input>
```

Here, the model can reliably identify the JSON inside `<input>` as data to be analyzed, not something to ignore or confuse with the request. The instructions outside the tag are clearly separate.

Counter-Example: If we just wrote: "Here is some JSON { ... }. Explain each field." without clear delimitation, the model might mistakenly blend the JSON into its answer or miss that it's supposed to focus only on that JSON.

Tip: Be consistent with whatever delimiters you choose. If you use `<example>...</example>` for examples, do so throughout your prompts. Consistent structure helps the AI parse your intent. You can even nest tags (e.g. `<examples><example>...`) to reflect hierarchy. Ultimately, a well-structured prompt is easier for both the model and you to debug.

Guiding Output Format and Style

Often you don't just need a correct answer you need it in a particular format or style. LLMs are quite capable of producing structured outputs on demand. The key is to tell them exactly what format you expect.

- **Specify the format explicitly:** If you want a JSON object, say "Respond with a JSON object containing the following keys...". If you need a bullet list or a table, mention that in the prompt. For complex formats, you might provide a template or schema. For example:
  - "Give the answer as a Markdown table with columns X, Y, Z."
  - "Output only valid JSON matching this schema: ..."
  - "Your response should be an SQL query that satisfies the request."

Clear formatting instructions ensure the model's response is directly usable without extensive editing. As Learn Prompting notes, without guidance, the AI might produce correct information in the wrong structure, requiring more work to extract or reformat.

If a format is critical (e.g. a snippet of code or data), it's wise to constrain the output: for instance, "Provide only the Python code, no explanation." or "Return the answer in exactly two sentences." LLMs will generally obey these constraints if clearly stated.

Controlling style is similarly done via instructions or role cues. You might add directives like "Explain in a friendly, conversational tone." or "Use technical language appropriate for an academic paper." Such style instructions can even be considered a part of the formatting requirement - they shape how the content is delivered, not just what the content is.

- **Multi-step approach:** Sometimes asking for reasoning and a formatted answer at once can produce conflated outputs. In these cases, you can prompt in stages. For example, first ask the model to think through the problem and silently arrive at an answer, then in a second prompt ask it to output the answer in a specific format (perhaps by feeding the reasoning back in a new prompt). This two-step "reason then format" approach can yield more reliable results when the format is complex (like code with specific structure, or answers with citations).

Example - Forcing a JSON Output:

> **User:** "Analyze the following text and output your findings as a JSON with fields `sentiment` and `key_points`. Text: <long text>" **Assistant:** `{"sentiment": "negative", "key_points": ["..."]}`

In this scenario, the user explicitly requested JSON and the assistant complied. The triple backticks help encapsulate the input text, and the model's training on JSON patterns lets it know how to format the output.

Bad Example: "Is the user message positive or negative? What are the key points in it?" (with no mention of format) - The model might respond in prose, e.g. "The message seems negative. Key points include...", which is correct content but not in JSON. This would require the developer to parse the text, an unnecessary complication.

In sum, treat the model like a flexible but literal-minded intern: if you need the report in Excel, tell them to use a table. If you need a sonnet, tell them the rhyme scheme. When you communicate the desired form clearly, the model will usually oblige.

Chain-of-Thought Reasoning

Complex problems often benefit from step-by-step reasoning, which can be encouraged through Chain-of-Thought (CoT) prompting. Instead of giving a one-shot answer, the model is prompted to "think out loud" - listing intermediate steps or an explanation before the final answer. This method yields more transparent and often more accurate results on tasks that involve multiple reasoning steps.

There are a few ways to invoke CoT:

- **Zero-Shot CoT:** Simply instruct the model to show its reasoning. A common phrase is "Let's think step by step." appended to the query. For example: "What is 19% of 5619? Let's think step by step." The model might then break down the calculation before giving the result.
- **Few-Shot CoT:** Provide examples of questions with step-by-step solutions in the prompt, then ask a new question. Seeing the pattern of reasoning in examples encourages the model to produce similar reasoning for the new query.
- **Structured CoT:** Use a format like: "First, explain your reasoning, then give the final answer on a new line." This explicitly separates the reasoning process from the answer.
- **Self-Consistency (Advanced):** One can generate multiple reasoning paths (via multiple outputs) and let the model choose the most common answer - an advanced technique useful in research, but less needed for everyday prompting.

Chain-of-thought helps in mathematical problems, logical reasoning, code analysis, and any scenario where a human would naturally break the task into sub-tasks. By letting the model "show its work," you also gain insight into where it might be going wrong, which aids debugging. In fact, Anthropic notes that seeing the model's thought process can highlight where a prompt was unclear, helping you refine it.

However, CoT does make the output longer (which can impact latency). Use it when needed, but not for trivial tasks. Also, not all models require an explicit CoT prompt some advanced models do complex reasoning internally. That said, even with GPT-4 or Claude 2, explicitly requesting reasoning can be useful if you want the rationale or if the problem is particularly tricky.

Example - CoT in action:

> **User:** "You have £100. You earn £30, then spend £20, then earn £50. How much do you have now? Let's work this out step by step." **Assistant:** "First, starting with £100. Earning £30 brings it to £130. Spending £20 reduces it to £110. Then earning another £50 brings it to £160. So the final amount is £160."

Here, the model followed the prompt to show each step, making it easy to verify the logic before accepting the answer. Many prompting guides emphasise: if a task is complex, let the model think. It's like asking a student to show their working - it generally leads to better outcomes

Few-Shot Prompting with Examples

Few-shot prompting (also known as "in-context learning") means including a few examples of the desired output behavior within your prompt. Instead of instructing the model from scratch, you show it: "When given input X, here's an ideal output Y." Providing 2-5 well-chosen examples can dramatically steer the model's responses

Key tips for effective few-shot prompting:

- **Relevance:** The examples should closely match the task and domain of your query. If you want the model to output an email draft, provide examples of email prompts and drafts.
- **Clarity:** Mark examples clearly, e.g. with a label like "Example 1 - Input:" and "Example 1 - Output:". Or use tags like `<example>` to wrap each example. This ties back to using structure to separate examples from the final query.
- **Consistency:** All examples should follow the same format or pattern you want in the output. The model may generalize from them. Inconsistency can confuse it.
- **Coverage:** Show diverse examples if possible, to cover different aspects of the task and avoid the model overfitting to one style. However, avoid contradictory examples.
- **Quantity:** You generally don't need many. Often 2-3 examples suffice. More can improve performance up to a point, but too many might consume context or cause the model to parrot the examples too closely.

Few-shot is especially useful for enforcing format or structure (the model "learns" the pattern from examples). It's also helpful for tasks with some complexity or creative element, where pure instructions might be interpreted in various ways. By giving examples, you nail down one particular way you want it done.

Example - Few-Shot Prompt: User prompt:

```
 You are a translator AI. Translate from English to French.
 Examples:
 English: "Good morning."
 French: "Bonjour."
 English: "How are you?"
 French: "Comment ça va ?"
 English: "I am learning French."
French:
```

Here we provided two English sentences with their French translations as examples. The model will infer the pattern and produce the French for "I am learning French." following the style of the examples (including

punctuation, etc.).

Why it works: The examples function as implicit training data the AI sees the translation mapping and continues it. This technique was famously shown to allow GPT-3 to translate, summarize, and perform tasks without explicit finetuning by just providing a few examples in the prompt.

One must be cautious to avoid prompt leakage where example outputs might contain information not intended in the final answer. Also, if examples are too dominant (e.g. 10 examples all doing the exact same thing), the model might become overly rigid or even copy them inappropriately. Balance and relevance are key.

In summary, show, don't just tell. High-quality examples can convey the nuances of format and tone better than instructions alone, and they can significantly improve accuracy, consistency, and performance on specific tasks.

Advanced Applications and Safeguards

Mitigating Hallucinations

One challenge with LLMs is their tendency to "hallucinate" - i.e. produce factually incorrect or made-up information in a confident manner. Even cutting-edge models can sometimes output inaccuracies or details not supported by any source. Prompt engineering offers some strategies to minimize this:

- **Provide Grounding Data:** Whenever possible, supply the model with relevant source text (documents, knowledge base snippets, etc.) in the prompt, and instruct it to only use that information for its answer. This is part of Retrieval-Augmented Generation (RAG). For example, "Using the information above, answer the question...". By grounding the response in provided context, you reduce off-base speculation.
- **Ask for Evidence or Quotes:** Encourage the model to cite sources or include direct quotes from the provided text. For instance, "Include the relevant quote and page number for each claim." This forces the model to align its answer with actual text. Anthropic suggests even having the model find supporting quotes after drafting an answer, and if none are found, revise the answer.
- **Allow Uncertainty:** Explicitly tell the model it's okay to say "I don't know" or "Not enough information" if it's unsure. This avoids the pressure to fabricate an answer. E.g. "If the answer is not in the passage, respond with 'I don't have that information." This simple instruction can drastically reduce false information.
- **Break the task down:** If a question is multi-faceted, use a chain-of-thought or multi-turn approach where the model can gather its thoughts. For example, first have it list what information is known vs unknown, then proceed to answer. This can prevent it from glossing over unknown parts.
- **Use Tools or Verification:** In critical cases, you can prompt the model to perform a web search (if the platform supports it), call a calculator, or otherwise verify facts with an external tool. Some frameworks allow you to set up "tool use" where the model output includes an action like `SEARCH("query")`, which is executed by an external system, then the results are fed back in. This can greatly help factual accuracy for up-to-date or detailed queries.
- **Higher Temperature for Brainstorm, Lower for Final:** A trick is to let the model be creative (higher temperature) when brainstorming possible answers or interpretations (to get diverse thoughts), then use a separate prompt with a low temperature to produce a final answer focusing only on the vetted information. This reduces out-of-left-field content.

Example - Grounded Answer:

> User provides an article about a medical study and asks: "According to the article, what were the two main findings?" **Better prompt:** "Read the article below and answer using only its content. Quote the sentences that state the results." This ensures the answer sticks to the article. The assistant might answer: "The article states, 'Finding A...' and 'Finding B...'. Therefore, the two main findings are A and B."

Bad prompt: "What were the main findings of the study?" (with the article text given but no instruction to use it) - The model might mix the article info with general knowledge or make something up if it's unsure.

No prompt can eliminate hallucinations entirely, but these techniques increase reliability. The goal is to channel the model's confidence into areas supported by data, and give it permission to admit ignorance when appropriate. As Anthropic's guide says, ensuring outputs are auditable (via quotes or citations) makes them more trustworthy.

Building Complex Prompts

In real-world applications, prompts often need to combine multiple techniques described above. Complex prompts might involve setting a role, providing extensive context, giving an example, demanding a specific format, and so on, all in one go. Essentially, you are writing a small script for the AI to follow.

Let's consider different domains and what a complex prompt might entail:

The following table:

Domain	Likely Prompt Components
Chatbots (Q&A)	System persona (consistent friendly tone) + relevant context or knowledge base + conversation history + user question + format (if any) + maybe few-shot for style. Often retrieval (RAG) is used to supply current info.
Legal Analysis	Strong expert role (""You are a veteran lawyer..."") + retrieved statutes/cases (grounding data) + perhaps a multi-step reasoning (CoT) to analyze facts + demand for structured output (e.g., ""Conclusion"" and ""Rationale"" sections) for clarity.
Finance / Analytics	Possibly prompt chaining: first prompt to gather facts, second to perform analysis. Or a single prompt with an embedded procedure (numbered steps to follow). Likely requires JSON or tabular output for numbers. Domain-specific role (financial analyst) for terminology.
Coding Assistance	Include code context (user's code or error) clearly separated (in markdown or `<code>` tags) + an expert coder persona (""You are a senior Python developer"") + chain-of-thought for debugging (""Think through the code step by step"") + maybe a few-shot examples of inputs and outputs (if doing something like code translation). Ensure the output is only code or an explanation as needed (formatting).

These are just examples many tasks combine elements. In Anthropic's interactive tutorial (Chapter 9: Building Complex Prompts), they showcase industry use cases like chatbots, legal assistants, financial reports, and coding helpers. Each case demonstrates weaving multiple prompt engineering tools together for the best result.

Think of prompts as modular components: you can reuse a well-crafted role prompt across many queries, or maintain a standard format template for Q&A tasks. For instance, you might always include a few-shot example of how to output an SQL query, whenever you ask for database-related help. Or have a standard system prompt that sets the tone for support tickets.

Example - Complex Prompt for Coding (with mistakes explanation):

```
System: "You are a coding assistant who not only provides code but also
explains it. Always
start with a brief explanation, then provide the code in a Markdown block.
If the code might
be error-prone, warn at theend."
```

```
User: "The user's code (below) is giving an IndexError. Fix the bug.
def find_item(lst):
return lst[1]
print(find_item([]))
```

Result: The assistant first explains the issue (IndexError due to empty list), then shows a corrected code block, then warns about empty input. **Analysis:** This prompt combined a role, an instruction to explain + code format, user code provided in markdown, and a specific error scenario. The assistant's answer is rich and structured, exactly as needed for a coding help context.

The art of complex prompting is in balancing guidance and freedom. Too rigid, and the model might get constrained or confused (especially if it can't fulfill all instructions perfectly). Too loose, and it might wander. Start simple, then layer on complexity as you identify needs. Often a few iterations get you to a prompt that covers all bases.

Beyond a Single Prompt: Advanced Workflows

Prompt Chaining

Instead of one giant prompt trying to do everything, consider breaking a task into a sequence of prompts, feeding the output of one step into the next. This approach is known as prompt chaining. It allows the model to tackle complex tasks in manageable chunks, which can improve both performance and interpretability

Why chain prompts?

- - It helps with accuracy, as each subtask gets the model's full attention without distraction from other aspects. It's less likely to drop details.
- - It provides clarity, since each prompt can focus on a single aspect with clearer instructions. It adds traceability and easier debugging: if the final answer has an issue, you can pinpoint which step (prompt) went wrong.

When to chain:

- Use chaining for multi-step processes like: data extraction → analysis → summary, or outline → draft → refine. If you notice the model struggling or mixing up steps in one prompt, that's a sign you can split the task
- For example, generating a long report with citations could be chained as: (1) gather relevant quotes, (2) analyse and draw conclusions from quotes, (3) format the final report with quotes and commentary.

How to chain:

- Design each prompt to produce an output that feeds into the next. You might need to instruct the model to output in a specific interim format (like JSON or a list) so that the next prompt can easily use it. Some systems allow passing the raw text output directly; others might require you to manually insert it into the next prompt.
- Anthropic's guide suggests using consistent tags to pass info between chain steps, e.g. wrapping an answer in `<analysis>...</analysis>` so the next prompt can refer to `<analysis>` content.

A simple chained workflow could be: 1. Prompt 1: "Read the customer reviews and list the top 3 complaints mentioned." -> Model outputs a list of complaints. 2. Prompt 2: "Given the complaints: [list from Prompt 1], suggest one improvement for each." -> Model outputs improvements. This is easier for the model than doing it in one prompt ("Read reviews and list complaints and improvements..." might cause it to lose structure).

Chaining also shines in interactive applications or agentic setups, where the model may loop through thought → action → observation → thought → action, etc., which is essentially chaining with logic (as in ReAct frameworks).

Keep in mind that chaining can consume more API calls and latency. But for critical tasks, the reliability gains often outweigh that. Each link in the chain is an opportunity to verify or modify outputs before moving on (you can even have human or programmatic checks in between).

Augmenting LLMs with External Tools

The capabilities of an LLM can be extended enormously by allowing it to use external tools or APIs. This concept underlies AI agents. Rather than the model simply spewing text, you let it output a special format that triggers some action outside the model, then feed the result back in. By doing this, the AI can, for example, perform calculations, fetch real-time information, or query databases - things beyond its trained knowledge.

For instance, you might design the system such that if the model produces an output like `CALCULATOR[(0.19)*(5619)]`, an external calculator will actually compute that. The result (1067.61 in that case) can then be given back to the model to incorporate into the final answer.

Common tool integrations include:

- **Search engines or knowledge bases:** The model can ask to search the web or a database for information. The retrieved text is then appended to the prompt.
- **Calculators or code evaluators:** The model can request a calculation or run a piece of code (e.g., to solve a math problem or simulate something).
- **APIs:** The model can produce a JSON with an API call specification, which your system executes and returns the result. E.g., `get_stock_price("GOOG")` -> returns price -> model continues with that info.

- **File systems:** It could create or read files (in a constrained environment), useful for more complex multi-step tasks.

In a prompt, you might instruct the model about available tools and the format to use. For example: "You can use the following tools: Calculator, WebSearch. To use a tool, output ToolName[input]. When you need no more tools, just provide the final answer." The conversation then involves the model alternating between reasoning and tool calls.

Why use tools? It compensates for the model's weaknesses (like up-to-date knowledge, or precise arithmetic) by delegating to specialized systems. As the Learn Prompting guide notes, this allows the model to incorporate information beyond its training data, making it far more powerful and reliable on dynamic queries. It's an emerging area that effectively turns a static LLM into an interactive problem- solving agent.

Quick Example - Weather Agent:

> **User:** "Will it rain in Barcelona tomorrow?" **Assistant thinking (hidden):** It might decide to use a weather API tool. **Assistant action:** `WEATHER_API["Barcelona", "2025-10-26"]` - (The system sees this and calls an actual weather API, gets result e.g. "Forecast says light rain"). **Assistant observation (hidden):** "Forecast says light rain." **Assistant answer:** "Yes according to the latest forecast, expect light rain in Barcelona tomorrow."

In this hypothetical chain, the model knew it should fetch current info rather than guess. The tool use format and result integration were pre-defined by the system designers.

From a prompting perspective, enabling tool use means formatting the prompt to inform the model of tools and how to call them, and then being able to capture those calls. It moves beyond pure text generation to an action loop. While implementing this is more involved than basic prompting, it illustrates the upper end of what prompt engineering can achieve when combined with software design.

(Note: The specifics of tool use vary by platform e.g., OpenAI uses "functions" in their API, and Anthropic's Claude has an "Assistant with Tools" mode via the MCP. The principles remain similar: the model is prompted about functions and expected to output a JSON or bracketed call which the host system executes.)

Search and Retrieval (RAG)

We touched on RAG (Retrieval-Augmented Generation) earlier as a way to mitigate hallucinations. It's worth emphasizing as an advanced architecture: combining LLMs with a search or retrieval system can drastically improve the quality and accuracy of responses.

In a RAG setup, when the user asks something, the system first uses a search module (which could be a vector similarity search over documents, or a traditional keyword search, etc.) to fetch relevant text snippets. These snippets are then included in the prompt given to the LLM. The LLM's job becomes synthesizing an answer based on that retrieved content, rather than relying purely on its internal knowledge.

Prompt-wise, this often means your user prompt might look like:

```
[System:] You are an AI assistant...
[User:]
DOCUMENTS:
1. "Excerpt from article A..."
```

```
 2. "Excerpt from article B..."
 QUESTION: "User's question here..."
 Instructions: Answer using the above documents and cite the source number
for
each fact.
```

Here, DOCUMENTS: are retrieved pieces of text. The prompt instructs the model to use them and even cite them (e.g. "According to doc 1, ...").

The benefit is clear: you get grounded answers with sources. Indeed, the Claude documentation notes that their search result integration feature enables "web search-quality citations" in responses and is powerful for RAG applications needing accurate sourcing.

Key considerations for RAG prompts:

- **Quality of retrieval:** The prompt is only as good as the documents you provide. Make sure your search retrieves relevant passages (using good queries, proper indexing, etc.).
- **Chunking and IDs:** Break knowledge sources into reasonably sized chunks and assign them identifiers. That way the model can quote or refer to a specific chunk without confusion.
- **Prompt limit:** You might not include all retrieved text if it's too much. Often you take the top N passages. Summarizing or filtering may help if they're lengthy.
- **Cite or not:** Telling the model to cite sources (like "[1]" corresponding to document

1. not only provides the user with transparency, it also implicitly forces the model to stick to those sources (since it knows it should only state what source 1/2 say, etc.). This is a clever prompt hack to keep it in line.

- **Failure mode:** If nothing relevant is found, you might prompt the model to say "I don't have information on that" rather than guessing.

Anthropic's own Claude has a dedicated search result augmentation where you can supply documents in a structured format and it will naturally cite them. Regardless of implementation, the prompt design pattern is similar across platforms.

Example - RAG Prompt:

> After searching a knowledge base, you feed Claude: "Context: Doc1: Mars is the fourth planet from the Sun and has two moons (Phobos and Deimos). Doc2: Jupiter is the fifth planet and has 79 moons. Question: How many moons does Mars have? Provide the answer with a citation." The model should answer: "Mars has two moons." (citing Doc1). If asked about Jupiter, it would say 79 with a citation to Doc2. If asked about Earth (not in docs), ideally: "The context provided does not mention Earth's moons."

This way, the knowledge is extendable and updatable without retraining the model - you just update your database. It's an extremely practical and increasingly common technique.

In conclusion, retrieval+LLM hybrids leverage the best of both worlds: the expansive reasoning and language ability of LLMs, and the factual accuracy of a curated knowledge source. Designing prompts for these systems requires juggling the extra context, but as we saw, it mostly involves clear structuring and source attribution instructions.

Resources:

Two GPTs and meta-prompts are include for generating good prompts and for GPTs:

- **Prompt Generator:** https://chatgpt.com/g/g-68fdf99d3d408191802fa7746fa9dde6-prompt-enhancement-protocol
- **Prompt Evaluator:** Meta-promtps: in `meta-prompts` folder

WATERMARK: This document was created by Antonio Lobo-Santos - AI Coding Seminar

---

Footer: Document prepared by Antonio Lobo-Santos for the AI Coding Seminar.

References:

1. **Anthropic (2025).** *Prompt Engineering – Overview.*
   Official Claude documentation covering prompt engineering best practices including clarity, examples, chain-of-thought reasoning, and role prompting.
   Available at: https://docs.claude.com/en/docs/build-with-claude/prompt-engineering/overview

2. **Anthropic (2024).** *Prompt Engineering Interactive Tutorial.*
   A multi-chapter GitHub tutorial illustrating basic to advanced prompt engineering concepts with examples.
   Available at: https://github.com/anthropics/prompt-eng-interactive-tutorial/tree/master

3. **Learn Prompting (2025).** *Prompt Engineering Guide: Introduction.*
   An open-source learning resource explaining prompt design principles, structure, and use of examples.
   Available at: https://learnprompting.org/docs/introduction