ChatGPT

Comprehensive Prompt Engineering Guide for LLMs

Prompt engineering is the craft of designing inputs (prompts) to guide large language models (LLMs) like GPT-4 towards high-quality outputs. Well-designed prompts can dramatically improve LLM behavior, enabling tasks from summarization and classification to complex, multi-step reasoning and tool use. This guide covers major prompting techniques (zero-shot, few-shot, chain-of-thought, function-calling, retrieval-augmented, and agentic prompting) with detailed best practices, annotated examples across domains, and strategies for evaluation and deployment. It is intended as an in-depth reference for AI practitioners building production systems.

Prompting Techniques

Zero-Shot Prompting

Zero-shot prompts ask the model to perform a task without giving any examples. The prompt itself contains all instructions and context needed. Best practices include **clear, specific instructions** and explicit formatting guidance. For example, instead of a vague command, specify format and content:

- **Be explicit and detailed.** Say exactly what you want (e.g. style, format, length) [1] [2].
- **Use delimiters or labels.** Prefix with `"""` or `###` to separate instructions from input text [3].
- **State output format.** Use phrases like "*Answer as a list of bullet points*" to constrain responses [2].
- **Provide context if needed.** Even without examples, include all relevant background in the prompt.

Example (Zero-Shot Summarization):

```
Summarize the following article in bullet-point form, focusing on key outcomes
and conclusions.

Article:
"""
Global research on climate change indicates a steady rise in average
temperatures...
(Actual article text)
"""

Expected Output:
- Average global temperatures have steadily increased...
- 2024 saw record-breaking heat waves...
- Main conclusion: urgent emission reductions are needed...
```

In this template, the instruction is explicit and separated from the content. The model is told to produce *bullet points*, preventing run-on answers [2]. The placeholder `{article_text}` can be replaced with any news or document text.

Example (Zero-Shot Classification):

```
Task: Classify the sentiment of the review below as Positive or Negative.

Review: "{review_text}"
Sentiment:
```

- *Expected Output (example)*: `Positive` or `Negative` (whichever applies to `{review_text}`).

Here we explicitly state the task and desired output label. By labeling the fields ("Task:", "Review:", "Sentiment:"), the model knows exactly what to do. This follows the principle of stating instructions clearly at the beginning [3].

Few-Shot Prompting

Few-shot prompts include a handful of **input–output examples** (demonstrations) to teach the model the task format. This often yields higher accuracy, especially for complex or ill-defined tasks. Key guidelines:

- **Choose high-quality, representative examples.** Examples should closely match the task domain and illustrate the desired format and style. For instance, if you want formal summaries, the examples should be formal.
- **Limit to a few examples.** Too many can exhaust context length; a handful (2–5) usually suffices [4].
- **Format clearly.** Use consistent formatting and separators (e.g. "Text 1: ..., Summary 1: ...") [5].
- **Order examples logically.** If relevant, sort by difficulty or category; or use random order to avoid bias [6].

Example (Few-Shot Classification):

```
Text 1: "I love this product, it works great!"
Sentiment 1: Positive

Text 2: "The service was terrible and very slow."
Sentiment 2: Negative

Text 3: "{new_review}"
Sentiment 3:
```

- *Expected Output (example)*: `Positive` or `Negative`.

This template shows two examples and then the new query. The model learns from Patterns (e.g. enthusiastic words = Positive) and applies to `{new_review}` [4]. Ensure each example is correctly answered; errors in examples will mislead the model.

Example (Few-Shot QA/Extraction):

```
Article: "TechCorp launched a new AI model named X in 2025. The model can
perform translations and summarization."
Query: "What is the name of the new model launched by TechCorp?"
Answer: "X"

Article: "{document_text}"
Query: "{user_question}"
Answer:
```

- *Expected Output (example)*: The answer extracted from `{document_text}`.

By giving the model an example of reading content and answering, it infers to follow the pattern. Note the use of clear labels ("Article:", "Query:", "Answer:"). Few-shot prompting aligns the model's output with the examples' style and detail level [4].

Chain-of-Thought (CoT) Prompting

Chain-of-thought prompting elicits the model's internal reasoning steps before giving a final answer [7]. This is especially effective for multi-step problems (math, logic puzzles, or any complex reasoning). There are two main approaches:

- **Zero-shot CoT:** Ask the model to "think aloud" by adding a cue such as *"Let's think step by step."* before the answer [8]. This prompts the LLM to generate intermediate reasoning.
- **Few-shot CoT:** Provide examples that include the reasoning chain. For example, show a question with a detailed "Answer:" that includes step-by-step calculations [9].

Key tips:

- **Explicitly request the chain.** Phrases like "*Explain your reasoning step by step, then give the answer*" can improve accuracy on complex tasks [8].
- **Use large models for CoT.** Chain-of-thought is most beneficial with very capable models (e.g. GPT-4-sized) [7]. Simple tasks or small models often don't see much gain.
- **Provide partial steps if needed.** Few-shot CoT can include blank spaces or "Answer:" tokens to encourage the model to fill in steps.

Example (Few-Shot CoT for Math):

```
Question: "If Alice has 5 books and gives 2 to Bob, how many does she have
left?"
Answer: She starts with 5 books. She gives 2 to Bob, so we calculate 5 - 2 =
```

```
<<5-2=3>>3. The answer is 3.

Question: "A train travels 60 miles per hour for 2 hours. How far does it
travel?"
Answer: The train speed is 60 miles/hour and it travels for 2 hours. So distance
= 60 * 2 = <<60*2=120>>120 miles. The answer is 120.
```

Here each "Answer:" includes arithmetic steps (enclosed in `<< >>` for clarity). The final answer follows "The answer is ...". When given a new question, the model will mimic this format.

Example (Zero-Shot CoT):

```
Question: "Marty has 100 cm of ribbon that he must cut into 4 equal parts, then
each part into 5 equal pieces. How long is each final piece?"
Answer: Let's think step by step. First, cutting 100 cm into 4 equal parts
yields 100 / 4 = <<100/4=25>>25 cm per part. Then each 25 cm part is cut into 5
equal pieces: 25 / 5 = <<25/5=5>>5 cm each. Therefore, each final piece is 5 cm
long.
```

The prompt "Let's think step by step" triggers detailed calculation before the conclusion, illustrating Zero-shot CoT [8]. The chain helps reduce mistakes in multi-step reasoning.

Function Calling (Tool Use)

Function calling is a new paradigm where the LLM can output JSON to invoke external APIs or tools [10]. This lets the model handle tasks beyond its static knowledge, like fetching live data or performing precise computations. For example, GPT-4 can output a function name and arguments instead of a plain text answer, which your application then executes. Key points:

- **Define tools in the system prompt or API call.** Provide the model with a schema and descriptions of functions it can call. GPT-4 (and GPT-4o) are trained to detect when a function should be used [11] [10].
- **Use JSON output.** The model will output something like:

  ```
  {"name": "get_current_weather", "arguments": {"location": "Paris", "unit":
  "celsius"}}
  ```

- **Chain calls if needed.** The model can call one function, get a result, and then continue reasoning (see the **Agentic Workflows** section).
- **Be explicit and unambiguous.** Ensure function names and parameters are clearly described. Good naming and descriptions help the model choose the right function [12].

Example (Weather Function Calling):

```
User: "What's the weather like in {city}?"
Assistant (function call): {
  "name": "get_current_weather",
  "arguments": {
    "location": "{city}",
    "unit": "celsius"
  }
}
```

Here the assistant responds with a JSON object specifying a function to call (`get_current_weather`). Once the function is executed by your system (returning, say, `{"temperature": 15,"description":"rainy"}`), the assistant can then generate a natural-language response. This connects LLM reasoning to real data [10] [11].

Retrieval-Augmented Generation (RAG)

Retrieval-augmented generation (RAG) enriches LLMs by providing them with external, up-to-date knowledge. In a RAG setup, a separate retrieval step finds relevant documents or database entries, and those are included in the prompt. The LLM then generates answers grounded in that context [13] [14]. RAG is crucial for:

- **Answering queries about recent or niche information.** For example, historical events, company internal docs, or any data beyond the model's training cutoff [13].
- **Reducing hallucinations.** By supplying factual content, the model is "anchored" to real information [15].
- **Domain-specific Q&A.** You can build expert systems by indexing domain texts (e.g. legal, medical) and using RAG to retrieve relevant snippets.

Typical RAG Workflow: Search → Retrieve documents → Prompt with context.

Figure: Example RAG pipeline. The system takes a query, retrieves relevant documents from a knowledge base, inserts them into the prompt, and the LLM generates a grounded answer.

Example (RAG for QA):

```
Query: "When was OpenAI founded?"
Retrieved Document: "OpenAI was founded in December 2015 by a group including
Elon Musk and Sam Altman..."
Answer:
OpenAI was founded in December 2015.
```

In practice, the LLM would see something like:

```
Context: "OpenAI was founded in December 2015..."
Question: "When was OpenAI founded?"
Answer:
```

and would complete “December 2015”. The retrieval component provided the exact fact. LangChain explains RAG as combining LLMs with external knowledge bases to overcome model limitations [14].

Agentic Prompting (Multi-Step/Tool Agents)

Agent-based prompting involves instructing the LLM to act as an **autonomous agent** that can plan, call tools, and iterate until the task is solved. This goes beyond a single turn: the model reasons, possibly invokes tools (via function calling), reflects on intermediate results, and continues. Key practices:

- **Use clear system instructions** to set the agent’s behavior. For example, include prompts emphasizing persistence (“*Only stop when the problem is solved*”) and tool usage (“*If unsure, use your tools – do not guess*”) [16].
- **Provide relevant tools/functions.** Define available functions (APIs) in the system prompt or API call. Name and describe them clearly; examples in the system prompt can help [12].
- **Encourage planning.** Optionally instruct the agent to outline a plan before acting on tools [17]. This improves reasoning: e.g. “*Plan extensively before each tool call and reflect on outcomes*” [17].
- **Maintain role consistency.** Agent prompts should make it explicit that the model is in charge of driving the solution, not just answering the user like a simple chatbot.

Example (Agent System Prompt):

```
System: You are an agent. Persistence: keep going until the user’s request is
fully resolved; do not stop midway. Tool usage: if you're unsure about
something, use the provided tools (do not guess or hallucinate). Planning: plan
your steps before calling any tool.
```

Then a user request might follow. For example:

```
User: "Help me organize these travel plans: find the cheapest flights to Paris
in June and suggest a 3-day itinerary."
```

The agent (GPT-4) would follow the instructions: possibly break the task into steps (find flights, then plan itinerary), call a flight-search API and a travel-guide API in sequence, and combine the results into a final answer. This mimics how LangChain defines agents: *“Agents are systems that take a high-level task and use an LLM as a reasoning engine to decide what actions to take and execute those actions.”* [18].

GPT-4.1’s prompting guide reports that these reminders (persistence, tool-calling, planning) significantly improved the model’s performance on coding tasks [16] [17]. In practice, you would structure the conversation with a strong system prompt (as above) followed by the user query and possibly ongoing multi-turn dialog where the model calls functions and gets results back.

Prompt Design and Debugging

Designing an effective prompt is an iterative process. Follow these steps:

1. **Define the goal precisely.** Know exactly what output format and content you need. This informs every prompt design choice.
2. **Start with a simple prompt.** Use basic instructions and test the model's response. Often "good enough" results emerge with just clear language [3] .
3. **Analyze errors or shortfalls.** If output is irrelevant, incomplete, or misformatted, identify why. Common issues include ambiguity in the task, no example guiding format, or missing context.
4. **Iterate and refine:**
5. **Add examples or context:** If the model misunderstands the task, consider adding a few demonstrations (few-shot) or more explicit instructions.
6. **Clarify format:** If parsing or structure is a problem, include a specific output template or mention how to format answers (e.g. "List: ...").
7. **Adjust tone/role:** If style or tone is off, incorporate role-playing (e.g. "Answer as a helpful tutor" or "In professional tone, ...").
8. **Handle negatives positively:** Rather than only saying "Don't do X" (negative instructions), explicitly state what to do [19] . For example, instead of "Do not mention unrelated facts," say "Mention only facts from the text provided."
9. **Use leading words or anchors:** For code or list outputs, using comment starters or specific labels can guide the structure [20] .

Illustrative Walkthrough: Consider designing a prompt for extracting keywords from text. A first attempt:

```
Prompt A (not fully effective):
"Extract the keywords from the following text: {text}"
```

This may give an unstructured sentence or miss key terms. Improve it by specifying the format and giving an example:

```
Prompt B (improved):
"Extract keywords from the text below and present them as a comma-separated
list.

Text: \"{text}\"

Keywords:"
```

Here we clarified the output format. Testing might still show issues (e.g. including stopwords). Further refine:

```
Prompt C (further refined):
"Identify the most important keywords in the text below. Provide exactly 5
```

```
keywords in a comma-separated list. Avoid very common words (stopwords).

Text: \"{text}\"

Keywords:"
```

By iteratively testing and adding constraints (“5 keywords”, “no stopwords”), the prompt becomes precise. This mirrors advice from the OpenAI docs to reduce vagueness and set exact criteria [21] .

Common Errors and Fixes:
- **Hallucination/inaccuracy:** If the model adds made-up details, add grounding context or use retrieval. In prompts without external context, explicitly **state uncertain areas** should be answered with “Unknown” or skipped.
- **Format violations:** If the model ignores bullet/list format, reinforce by showing format in prompt or giving a mini-example [22] [23] .
- **Overly long/verbose answers:** Constrain length (“in 3 sentences”, “50-word summary”) [21] .
- **Unwanted behavior:** Instead of “Don’t do X”, use “Do Y instead” [19] to positively guide behavior.

Throughout, maintain concise prompts and test with various inputs. Track changes (see Prompt Management section) so you can revert or analyze versions.

Evaluating Prompt Effectiveness

Evaluating prompts ensures they meet task requirements reliably. Use both **manual review** and **automated metrics**:

- **Manual testing:** Run diverse test cases through the prompt and inspect outputs. Cover edge cases, varied phrasing, and difficulty levels. Have domain experts review for accuracy and relevance.
- **Automated metrics:** Depending on the task, use appropriate metrics: accuracy or F1 for classification, BLEU/ROUGE for summarization, BLEU/EM for translation, etc. For tasks without ground truth, consider semantic similarity or LLM-based scoring.
- **Reference-based evaluation:** When you have a ground-truth answer, compare the model’s output to it [24] . For example, compute exact-match ratio for answers or use text overlap measures.
- **Reference-free evaluation:** Use proxies like output length, presence of forbidden content, or even a second LLM as a judge [25] . For instance, you can prompt GPT-4 itself to critique an output (“Rate the factuality of this answer on a scale of 1-5”). LLM-as-judge is a common technique [24] .
- **Task-specific checks:** For factual tasks, cross-check facts; for code, run tests; for data extraction, verify schema.

Document all prompt versions, inputs, outputs, and evaluation results. As recommended in LLM Ops practice, log prompts and responses along with model version and settings [26] . This enables trend analysis over time and helps identify when a prompt breaks (e.g. after a model update).

Prompt Management and Integration

In production systems, prompts should be treated like code. Best practices include:

- **Modularize and template.** Break prompts into reusable templates with placeholders (variables). Frameworks like LangChain's `PromptTemplate` and `ChatPromptTemplate` let you define prompts with slots for dynamic content [27] [28]. This ensures consistency and easier maintenance. For example:

  ```
  from langchain_core.prompts import ChatPromptTemplate
  prompt = ChatPromptTemplate.from_template([
      ("system", "You are a helpful summarizer."),
      ("user", "Summarize the following text:\n\n{text}")
  ])
  result = prompt.invoke({"text": article_text})
  ```

  This separates the fixed instruction ("You are a helpful summarizer.") from the variable user input, making updates straightforward [27] [28].

- **Version control.** Store prompts (or template code) in version control (git) to track changes [29]. A simple change log of prompt edits aids rollback and debugging of regressions. PromptHub-like tools can also manage prompt versions and A/B test variants.

- **Decouple from application logic.** Keep prompts in a separate repository or service rather than embedding as long strings in code [30]. This allows non-developers (e.g. prompt engineers or product managers) to view and edit prompts without touching code.
- **Monitor usage and cost.** Track prompt lengths, token usage, and frequency. Long prompts and verbose outputs increase costs [31]. Analyze logs to identify unusually long generations or frequent completions.

- **Collaboration and access control.** Use tools (PromptFlow, LangFlow, PromptLayer, etc.) or internal dashboards to allow team members to test prompts and see results [32]. The Qwak blog notes that prompt management tools enable different stakeholders (developers, domain experts) to deploy prompts independently [33].

- **Consistency across models.** A prompt that works well on one model may not perform the same on another [26]. When upgrading model versions (e.g. GPT-3.5 → GPT-4), re-run your evaluation suite. You may need to tweak wording due to differences in instruction following [34].

In summary, treat prompts as first-class assets: modular, versioned, and continuously evaluated. Integrate them via libraries (LangChain, LlamaIndex, etc.) or custom code that can fill in templates and invoke the OpenAI (or other) API.

Use Case Examples

Below are illustrative use cases demonstrating how prompt techniques apply in different domains. Each example includes a prompt template (with placeholders) and a sample input/output.

Summarization (Business/News)

Professionals often need concise summaries of long texts. Best practices: ask for specific formats (bullets, tweets, etc.), set length constraints, and provide role or audience context if needed [35] [2].

- **Few-shot strategy:** Show one or two summarized examples to teach style.
- **Zero-shot + instructions:** Provide clear formatting rules.

Template Example:

```
Task: Provide a bullet-point summary.
Article: "{article_text}"
Summary (bullet points):
-
```

- *Sample Input:*
Article: "Acme Corp's Q1 report shows revenue grew by 15% over last year, driven by increased sales in Europe. CEO plans to expand the product line next quarter..."
- *Sample Output:*
- "Acme Corp Q1 revenue up 15% YoY (driven by European sales)."
- "CEO announces product line expansion next quarter."
- "Strong financial performance attributed to market demand and efficiency gains."

This prompt explicitly says "bullet-point summary" and prefaces the output with a dash. It may help to also instruct style ("concise, in 5 bullets"). Role-based cues ("As an analyst..") can tailor tone (PromptLayer suggests e.g. writing for a specific audience) [35].

Classification (NLP/Data Tagging)

Classification tasks (sentiment analysis, spam detection, topic labeling) can often be done in zero-shot or few-shot style. Provide clear labels and criteria.

- **Zero-shot approach:** Clearly state categories and format. E.g. "Answer exactly 'Positive' or 'Negative'."
- **Few-shot approach:** Provide a few labeled examples to define the pattern.

Template Example:

```
Task: Classify the following review as Spam or Not Spam.
Review: "{review_text}"
Classification:
```

- *Sample Input:* `Review: "Limited time offer!!! Click here to claim your prize!"`
- *Sample Output:* `Spam`

By specifying the labels upfront and including examples in the prompt, the model learns the decision criteria. The OpenAI guide suggests starting with zero-shot, then moving to few-shot if needed [4].

Customer Service / Chatbot (Support Dialogue)

For chatbots and service agents, prompts often define roles and conversation format. Use system instructions like **"You are an empathetic support agent"**. Chain-of-thought is less common here; clarity and politeness are key.

- **Use role-play:** E.g. "*As a travel agent, help the customer...*" can guide style.
- **Few-shot dialogues:** Include one example turn or conversation snippet to set tone.

Template Example:

```
System: "You are a helpful customer support agent. Respond helpfully and
politely. Do not ask for personal data like passwords."

User: "{user_message}"
Agent:
```

- *Sample Input:* `User: "I forgot my account password and can't log in."`
- *Sample Output:*

`Agent: "I'm sorry to hear that. You can reset your password by clicking 'Forgot Password' on the login page. Would you like me to guide you through the steps?"`

Notice the system message sets constraints ("don't ask for PII") and tone. The user message is inserted, and the assistant's response should follow. OpenAI's advice is to tell *what to do* (e.g. "refer to help article") instead of "don't do X" [19], which is reflected here by phrasing policies positively.

Data Analysis / Coding

When using LLMs for data tasks or code generation, chain-of-thought and function-calling can help. For example, to generate SQL queries or analyze data:

- **CoT for complex logic:** "*Explain your reasoning step by step and then show the code.*"
- **Function for actual execution:** Combine LLM with Python or SQL via function-calling (tools like Python REPL).

Template Example (Code Generation):

```
Task: Write a Python function to calculate the factorial of a number n.

Assistant:
```python
def factorial(n):
    ...
```

- *Sample Output:* (A complete Python function with correct logic)

Using code-block formatting and prompting with comments or code context helps the model output syntactically correct code. The OpenAI API guide suggests using comments or "leading words" to nudge patterns [20] (see how the prompt starts with a Python comment).

Education / Tutoring

In educational settings, prompts might require the model to adopt a teaching role. Techniques:

- **Role-playing:** "*Explain as if teaching a high school student.*" (PromptLayer highlights using audience roles [35]).
- **Few-shot quiz:** Provide example question-answer pairs to set format.
- **Chain-of-Thought:** Encourage step-by-step reasoning for problem-solving.

Template Example:

```
You are a math tutor. Explain the following problem step by step:
Problem: "{math_problem}"
Solution:
```

- *Sample Input:* `Problem: "If x + 3 = 7, what is x?"`
- *Sample Output:* `Solution: First, I subtract 3 from both sides: x = 7 - 3. So x = 4.`

This prompt clearly assigns a role ("math tutor") and expects an explanation. The solution is phrased as teaching ("First, I subtract..."). Providing simple example Q&A in the prompt can further align style and thoroughness.

Summary of Best Practices

- **Be Specific:** Use detailed instructions, examples, and format cues [3].
- **Iterate Quickly:** Test prompts, identify issues, and refine (the empirical nature of prompt engineering) [36] [37].
- **Use Examples:** When zero-shot fails, add few-shot examples to show the model exactly what output you expect [4].
```

- **Encourage Thinking:** For hard problems, prompt step-by-step reasoning (CoT) or iterative tool use [7] [16].
- **Constrain and Format:** Explicitly state output formats or constraints to avoid ambiguity [22] [14].
- **Leverage Tools:** Use function calling and retrieval to overcome the model's limitations and keep answers grounded [10] [13].
- **Monitor and Evaluate:** Continuously validate outputs with metrics and human review [24] [26].
- **Manage Prompts as Code:** Version, modularize, and log prompts for reliable production use [29] [27].

By following these practices and tailoring prompts to your domain, you can harness GPT-style models effectively in real-world applications. Remember that prompt engineering is an iterative, empirical craft: always test, measure, and improve your prompts based on feedback and evaluation results [37] [24].

---

[1] [2] [3] [4] [5] [19] [20] [21] [22] [23] [36] Best practices for prompt engineering with the OpenAI API | OpenAI Help Center
https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api

[6] [7] [8] [9] [13] Prompt Engineering | Lil'Log
https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/

[10] Function Calling with LLMs | Prompt Engineering Guide
https://www.promptingguide.ai/applications/function_calling

[11] [12] [16] [17] [34] GPT-4.1 Prompting Guide
https://cookbook.openai.com/examples/gpt4-1_prompting_guide

[14] [15] Retrieval augmented generation (RAG) | LangChain
https://python.langchain.com/docs/concepts/rag/

[18] Agents | LangChain
https://python.langchain.com/docs/concepts/agents/

[24] [25] LLM evaluation metrics and methods
https://www.evidentlyai.com/llm-guide/llm-evaluation-metrics

[26] [29] [30] [31] [32] [33] What is Prompt Management? Tools, Tips and Best Practices | JFrog ML
https://www.qwak.com/post/prompt-management

[27] [28] Prompt Templates | LangChain
https://python.langchain.com/docs/concepts/prompt_templates/

[35] Best Prompts Asking for a Summary: Top AI Techniques
https://blog.promptlayer.com/best-prompts-for-asking-a-summary-a-guide-to-effective-ai-summarization/

[37] Prompt engineering: A guide to improving LLM performance | CircleCI
https://circleci.com/blog/prompt-engineering/