Prompt Engineering

Techniques for Effective Communication with Large Language Models

February 2026 • Technical Report

Table of Contents

1. Introduction

Prompt engineering is the practice of designing and refining input text to elicit desired behaviors and outputs from large language models. As LLMs have become general-purpose reasoning and generation engines, the quality of the prompt has emerged as one of the most significant determinants of output quality. Unlike traditional software where behavior is specified through code, LLMs are steered through natural language instructions, examples, and structural cues embedded in the prompt.

This report surveys the major prompt engineering techniques, from foundational methods like few-shot prompting and chain-of-thought reasoning to advanced patterns for structured output generation, tool use, and system prompt design. The focus is on practical, empirically validated techniques applicable to modern instruction-tuned models.

2. Foundational Techniques

2.1 Zero-Shot Prompting

Zero-shot prompting provides the model with a task description and input but no examples of the desired output. Modern instruction-tuned models perform well in zero-shot settings for straightforward tasks like summarization, translation, and simple question answering. The effectiveness of zero-shot prompting depends heavily on the clarity and specificity of the instruction. Vague instructions like "analyze this text" produce less useful results than specific ones like "identify the three main arguments in this text and evaluate the evidence supporting each one."

2.2 Few-Shot Prompting

Few-shot prompting includes one or more examples of input-output pairs before the actual query, demonstrating the desired format, style, and reasoning pattern. Few-shot examples are particularly valuable for tasks with specific output formats, domain-specific conventions, or subtle quality criteria that are difficult to express in natural language instructions alone. Research has shown that example selection, ordering, and diversity all affect performance, and that examples drawn from a distribution similar to the target task yield the best results.

2.3 Chain-of-Thought (CoT) Prompting

Chain-of-thought prompting instructs the model to show its reasoning process step by step before producing a final answer. This technique dramatically improves performance on tasks requiring multi-step reasoning, including mathematical problem solving, logical deduction, and complex analysis. CoT can be elicited with simple instructions like "think step by step" (zero-shot CoT) or by providing examples that include explicit reasoning traces (few-shot CoT). The reasoning process helps the

model avoid common errors by forcing it to decompose complex problems into manageable sub-steps.

3. Advanced Reasoning Techniques

3.1 Self-Consistency

Self-consistency extends chain-of-thought prompting by sampling multiple independent reasoning paths and selecting the most common final answer through majority voting. This approach exploits the observation that correct reasoning paths tend to converge on the same answer while incorrect paths diverge. Self-consistency is particularly effective for mathematical and logical reasoning tasks and can significantly improve accuracy at the cost of increased token usage.

3.2 Tree-of-Thought and Graph-of-Thought

Tree-of-thought prompting structures the reasoning process as an explicit search tree, where the model generates multiple candidate next steps at each reasoning stage and evaluates which branches are most promising before continuing. This allows exploration of different solution strategies and backtracking from dead ends. Graph-of-thought extends this further by allowing reasoning nodes to merge and reference each other, supporting more complex reasoning topologies. These techniques are most valuable for planning, strategy, and open-ended problem solving.

3.3 Retrieval-Augmented Generation (RAG)

RAG augments the prompt with relevant information retrieved from external knowledge sources based on the user's query. A retrieval system (typically using embedding-based similarity search) identifies relevant documents or passages, which are then included in the prompt as context for the model to reference. RAG reduces hallucination by grounding responses in specific sources, enables models to access information beyond their training data, and provides natural citation support. Effective RAG systems require attention to chunk size, retrieval quality, context window management, and instruction formatting that directs the model to use the provided context.

4. Structured Output and Control

4.1 Output Format Specification

Many applications require model outputs in specific formats such as JSON, XML, CSV, or custom schemas. Effective format specification combines explicit instructions ("respond in valid JSON matching this schema"), few-shot examples showing the exact format, and structural cues like opening brackets or tags that prime the model to continue in the specified format. Modern serving frameworks support constrained

decoding that guarantees syntactically valid structured output by masking invalid tokens during generation, eliminating parsing failures entirely.

4.2 System Prompts

System prompts establish the model's role, personality, constraints, and behavioral guidelines for an entire conversation. Effective system prompts define the assistant's expertise and scope, specify response style and format preferences, establish safety boundaries and refusal criteria, and provide domain-specific context. System prompts are processed once during prefill and their KV cache is reused across conversation turns, making them computationally efficient for multi-turn interactions. Well-designed system prompts can dramatically shape model behavior without requiring fine-tuning.

4.3 Tool Use and Function Calling

Modern LLMs support tool use through prompting patterns that teach the model to recognize when external tools are needed, generate structured tool calls with appropriate parameters, interpret tool results, and incorporate them into the response. Tool use extends LLM capabilities to include real-time data access, computation, code execution, API integration, and interaction with external systems. Effective tool-use prompting requires clear descriptions of available tools, their parameters, and usage constraints, along with examples of multi-step tool use chains for complex workflows.

5. Prompt Optimization Strategies

5.1 Iterative Refinement

Prompt development is an iterative process. Starting with a simple prompt, the engineer examines failure modes on representative test cases and refines the prompt to address them. Common refinements include adding edge case instructions, providing negative examples showing what not to do, clarifying ambiguous terminology, and adjusting the level of detail in instructions. Maintaining a diverse test set that covers expected input variations is essential for systematic prompt improvement.

5.2 Prompt Decomposition

Complex tasks often benefit from being decomposed into a pipeline of simpler prompts, each handling one aspect of the problem. For example, a document analysis task might use separate prompts for extraction, classification, reasoning, and formatting. This modular approach allows each prompt to be optimized independently, makes debugging easier, and often produces better results than a single monolithic prompt attempting to handle all aspects simultaneously.

5.3 Automated Prompt Optimization

Automated prompt optimization methods use LLMs to improve prompts programmatically. DSPy provides a framework for defining LLM pipelines as programs with optimizable prompt components, automatically tuning demonstrations and instructions based on a metric and validation set. Other approaches include OPRO (using an LLM to propose prompt improvements) and gradient-based methods that optimize soft prompt embeddings. These automated approaches can discover non-obvious prompt formulations that outperform human-designed prompts.

6. Common Pitfalls and Best Practices

Pitfall	Impact	Best Practice
Vague instructions	Inconsistent, low-quality output	Be specific about format, length, and criteria
No examples for complex formats	Format errors, parsing failures	Include 2–3 diverse few-shot examples
Ignoring reasoning	Errors on multi-step tasks	Request step-by-step reasoning (CoT)
Prompt too long	Key info lost in middle	Front-load critical instructions; trim context
No negative examples	Unwanted behaviors persist	Show what to avoid, not just what to do
Static prompts for dynamic tasks	Brittle, fails on edge cases	Use RAG, tool use, or adaptive prompting

7. Model-Specific Considerations

Different model families respond differently to prompting techniques due to variations in training data, instruction tuning methodology, and architecture. Techniques that work well for one model may be less effective for another. System prompt formats, special tokens, and tool-use protocols vary across providers. Practically, prompt engineers should test their prompts against the specific model being deployed and avoid assuming that prompts transfer perfectly across model families. The trend toward standardized chat templates and tool-calling APIs is gradually reducing these compatibility concerns.

8. Future Directions

Prompt engineering is evolving from a manual craft toward a more systematic discipline. The development of prompt programming frameworks like DSPy signals a shift toward treating prompts as optimizable components in larger systems. Multi-modal prompting, incorporating images, audio, and video alongside text, introduces new design

challenges and opportunities. As models become more capable at following complex instructions, the emphasis is shifting from coaxing correct behavior through clever prompt tricks toward clear, well-structured communication of intent, much like writing specifications for a capable human collaborator.

9. Conclusion

Prompt engineering is the primary interface through which practitioners unlock the capabilities of large language models. From simple instruction formulation to sophisticated reasoning chains, structured outputs, and tool-integrated workflows, the techniques covered in this report form the essential toolkit for anyone building applications on top of LLMs. While the field continues to evolve as models become more capable and prompt optimization becomes more automated, the fundamental skill of clearly communicating intent and constraints through well-crafted prompts remains central to effective LLM use.