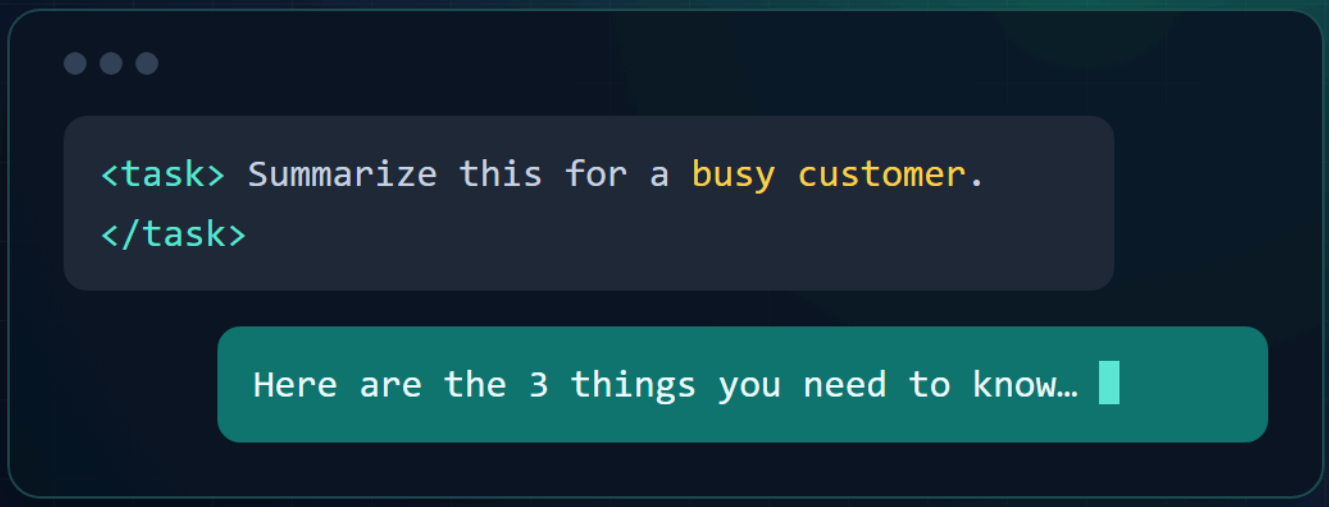


THE COMPLETE PRACTICAL GUIDE

Prompt Engineering

Instructing LLMs and AI Agents — from first prompt to production

TECHNIQUES · RAG · TOOLS · AGENTS · EVALS · SECURITY

THENEXBOOKS

Prompt Engineering

The Complete Practical Guide to Instructing LLMs and AI Agents

First Edition

THENEXBOOKS

Prompt Engineering
The Complete Practical Guide to Instructing LLMs and AI Agents

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Prompts and code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. Large language models change frequently, and their behavior varies between providers and versions; every technique in this book should be tested with the models you actually use. The publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Claude and Anthropic, GPT and OpenAI, Gemini and Google, Llama and Meta, Mistral, Qwen, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement.

Preface

Large language models can draft, summarize, translate, classify, write code, analyze data, and — when given tools — take actions in the world. Yet the same model that produces a brilliant answer for one person gives a vague, wrong, or unsafe answer to another. Very often the difference is not the model. It is the prompt.

Prompt engineering is the discipline of communicating with models clearly and reliably: saying what you want, giving the right context, showing examples, structuring inputs and outputs, and testing until the behavior is dependable. It is less about magic phrases and more about the same skills that make someone a great manager, teacher, or technical writer — plus an engineering mindset for measurement and iteration.

This book is a complete, practical guide. It starts with how models read your words and builds up to production systems: retrieval-augmented generation, tool use, agents, context engineering, evaluation, security, cost control, and operating prompts as a team. Every chapter is full of **before-and-after prompts** so you can see exactly what changes and why it works.

Who This Book Is For

- **Developers** building LLM features, RAG systems, and agents through APIs.
- **Product builders and founders** who need reliable AI behavior, not demos.
- **Analysts, writers, and operators** who use AI assistants every day and want consistently better results.
- **Team leads** who need shared practices for testing, securing, and maintaining prompts.

What You Will Learn

- How LLMs process prompts — tokens, context windows, sampling — and what that means for how you write.
- The building blocks of great prompts: clear instructions, examples, structure, roles, and output formats.
- Core techniques: reasoning, structured output, long-context grounding, chaining, and self-critique.
- Proven patterns for summarization, extraction, classification, writing, coding, SQL, and translation.
- How to build RAG systems, tool-using assistants, and agents — and how to engineer their context.
- How to evaluate, secure, speed up, and operate prompts in production.
- Two end-to-end case studies you can adapt to your own products.

How This Book Is Organized

Part I — Foundations (Chapters 1–6) explains how models work and how to write clear, well-structured prompts. **Part II — Core Techniques** (Chapters 7–12) covers reasoning, roles, structured output, long context, chaining, and self-critique. **Part III — Applications** (Chapters 13–18) applies these techniques to the most common real-world tasks. **Part IV — Retrieval, Tools, and Agents** (Chapters 19–22) moves from single prompts to systems that search, act, and plan. **Part V — Production** (Chapters 23–26) addresses evaluation, security, cost, and operations. **Part VI — Case Studies** (Chapters 27–28) brings everything together. The appendices provide a pattern catalog, a template library, a checklist, and a glossary.

A Note on Models and Versions

This book is deliberately **vendor-neutral**. The techniques apply to Claude, GPT, Gemini, and open-weight models alike, although each model family has its own strengths and quirks. Code examples use a placeholder model identifier (`MODEL = "your-model-id"`) so they stay valid as models evolve. Always consult your provider's current documentation, and always test.

The Running Example: Orbit Store

Throughout the book we build AI features for **Orbit Store**, a fictional e-commerce marketplace: a customer-support assistant, a product-description writer, a review summarizer, a ticket router, a catalog extractor for supplier PDFs, an analytics assistant for merchants, a shopping agent, and a merchant-operations agent.

Conventions Used in This Book

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake or a behavior that can cause wrong answers, data leaks, or unsafe actions.

Prompts and code appear in `monospace` blocks. "Before" and "after" labels mark prompt rewrites.

Contents

Preface

PART I — FOUNDATIONS

1 What Prompt Engineering Is (and Isn't)

2 How LLMs Work: Tokens, Context Windows, and Sampling

3 Anatomy of a Prompt: System, User, and Assistant Messages

4 Clarity and Specificity: Writing Instructions Models Can Follow

5 Learning from Examples: Zero-Shot, One-Shot, and Few-Shot

6 Structuring Prompts: Delimiters, XML Tags, and Templates

PART II — CORE TECHNIQUES

7 Reasoning: Chain-of-Thought and Reasoning Models

8 Roles, Personas, Tone, and Audience

9 Structured Output: JSON, Schemas, and Validation

10 Working with Long Context: Documents, Grounding, and Citations

11 Decomposition and Prompt Chaining

12 Self-Consistency, Self-Critique, and Reflection

PART III — APPLICATIONS

13 Summarization and Information Extraction

14 Classification, Routing, and Moderation

15 Writing, Editing, and Style Control

16 Code Generation, Review, and Debugging

17 Data Analysis and Text-to-SQL

18 Multilingual Prompting and Translation

PART IV — RETRIEVAL, TOOLS, AND AGENTS

19 Retrieval-Augmented Generation (RAG)

20 Tool Use and Function Calling

21 Building Agents: Loops, Planning, and Memory

22 Context Engineering for Agents (including MCP)

PART V — PRODUCTION

23 Evaluating Prompts: Test Sets, Metrics, and LLM-as-Judge

24 Prompt Security: Injection, Jailbreaks, and Guardrails

25 Cost, Latency, and Reliability

26 PromptOps: Versioning, Templates, Monitoring, and Iteration

PART VI — CASE STUDIES

27 Case Study: The Orbit Store Support Assistant

28 Case Study: An Agentic Catalog and Operations Pipeline

APPENDICES

A Prompt Pattern Catalog

B Template Library

C Production Checklist

D Glossary

PART I

Foundations

How models read your words, and how to write prompts that are clear, specific, well-structured, and backed by the right examples.

1 · What Prompt Engineering Is
2 · How LLMs Work
3 · Anatomy of a Prompt
4 · Clarity and Specificity
5 · Learning from Examples
6 · Structuring Prompts

CHAPTER 1 · PART I — FOUNDATIONS

What Prompt Engineering Is (and Isn't)

Prompt engineering is the practical discipline of telling a language model what you want so that it reliably does it. In this chapter we define the discipline precisely, separate it from the techniques it is often confused with, clear away the most persistent myths, and introduce Orbit Store, the fictional marketplace whose problems we will solve throughout the book.

A Working Definition

Every interaction with a large language model (LLM) begins with text that you supply. That text might be a single question typed into a chat window, a carefully versioned system prompt deployed to thousands of users, or a bundle of instructions, documents, and tool descriptions assembled by software at run time. Whatever its shape, that input is the prompt, and it is the main lever you have over what the model produces.

We will use this working definition throughout the book:

> **DEFINITION**
> Prompt engineering is the practice of designing, testing, and maintaining the inputs to a language model so that its outputs are useful, accurate, safe, and consistent for a specific task and audience.

Three words in that definition do most of the work. **Designing** means making deliberate choices: what the model needs to know, what it should do, what the output should look like, and what it must avoid. **Testing** means you do not trust a prompt because it reads well to you; you trust it because you ran it against realistic inputs and inspected the results. **Maintaining** means prompts are living artifacts. Models get upgraded, product requirements shift, and users find inputs you never imagined. A prompt that worked in March may quietly degrade by September.

Notice what the definition does not say. It does not say "finding the magic phrase that unlocks the model." It does not say "tricking the model." And it does not restrict itself to chat. Much of the most valuable prompt engineering happens in code, where no human ever reads the prompt except the engineers who wrote it.

Prompt engineering is communication, not incantation

The most useful mental shift for newcomers is to treat a prompt as a briefing for a highly capable collaborator who knows a great deal about the world but nothing about your situation. That collaborator has never seen your product, does not know your customers, has no idea which of three

plausible interpretations you meant, and cannot read your mind about format. Everything that makes a good briefing for a human contractor — a clear goal, relevant context, explicit constraints, an example of what "good" looks like — also makes a good prompt. Chapter 4 develops this idea into a full method.

The difference from human communication is that the model will almost never push back and ask what you meant. It will pick an interpretation and commit to it, fluently and confidently. That is why vague prompts are so dangerous: the output often *looks* right even when it answers a different question from the one you had in mind.

Prompt — before

```
Write a product description for the backpack.
```

Prompt — after

```
Write a product description for the Orbit Store listing below.

Audience: commuters who cycle to work and care about weather protection.
Length: 80-110 words, one paragraph, no bullet points.
Tone: practical and warm; avoid superlatives like "best" or "ultimate".
Only mention features that appear in the listing data.

Listing data:
- Name: Nimbus 22L Roll-Top Backpack
- Material: recycled nylon with TPU coating (waterproof)
- Laptop sleeve: fits up to 15 inches
- Reflective strips: front and side
- Weight: 780 g
```

The first prompt forces the model to guess at nearly everything: which backpack, for whom, how long, in what voice, and whether it may invent features. The second prompt answers those questions up front. Neither prompt contains any clever wording; the improvement comes entirely from supplying the information a competent copywriter would have asked for.

Why Prompts Still Matter as Models Improve

A common prediction is that prompt engineering is a temporary skill: models will become so capable that they will understand any request, however sloppy. There is some truth in the direction of that claim. Many current models are much better than their predecessors at inferring intent from short requests, following long instruction lists, and recovering from ambiguous phrasing. Some tricks that mattered a few years ago — elaborate role-play framings, repeated emphasis, specific incantations — matter less or not at all with many current models.

But the core of prompt engineering is not going away, for four reasons.

1. **Models cannot know what you have not told them.** No amount of intelligence lets a model deduce your refund policy, your brand voice, the JSON schema your downstream parser expects, or the fact that your finance team wants amounts in euros. Context has to be supplied, and deciding what context to supply is prompt engineering.
2. **Ambiguity is a property of the request, not the model.** "Summarize this ticket" has many correct answers. A one-line summary for a dashboard and a detailed handoff note for a senior agent are both summaries. A more capable model will produce a better summary of *some* kind; only you can say which kind you need.
3. **Production systems need consistency, not just capability.** A chat user can rephrase if the first answer misses. An application processing ten thousand support tickets a day cannot. Getting a model to behave the same way on the ten-thousandth input as on the first is an engineering problem, and prompts are a large part of the solution.
4. **The scope of prompting keeps growing.** As models gain tools, long context windows, and the ability to act as agents over many steps, the "prompt" expands to include tool descriptions, retrieved documents, memory, and the running history of an agent's work. The discipline of deciding what goes into the model's context — sometimes called *context engineering*, which we cover in Chapter 22 — is prompt engineering at a larger scale.

A better model raises the ceiling of what a good prompt can achieve. It does not remove the gap between a good prompt and a poor one; often it widens it, because a capable model follows a precise brief very precisely, and follows a vague brief in confident, unpredictable directions.

Prompt Engineering vs Fine-Tuning vs RAG vs Tool Use

When a model does not behave the way you want, you have several levers. Prompt engineering is usually the first one to pull, but it is not the only one, and knowing the alternatives helps you recognize when a prompt is the wrong tool.

- **Prompt engineering** changes the input at inference time. It is fast to iterate on, requires no training infrastructure, and works with any model you can call. Its limits: everything must fit in the context window, and each call pays for the tokens you send.
- **Fine-tuning** changes the model's weights by training it further on your examples. It is useful for teaching a consistent style or format across many calls, compressing a long prompt into learned behavior, or adapting a smaller model to a narrow task. It is slower to iterate on, requires a curated dataset, and is a poor way to teach the model *facts* that change often.
- **Retrieval-augmented generation (RAG)**, introduced in the research literature by Lewis et al. (2020), fetches relevant documents at query time and places them in the prompt. It is the standard approach when answers depend on large or frequently updated knowledge — product catalogs, policy documents, knowledge bases. Chapter 19 covers it in depth.

- **Tool use** (also called function calling) lets the model request that your code run an action — look up an order, query a database, send an email — and then continue with the result. It is the right choice when the model needs live data or must take actions in the world. Chapter 20 covers it.

These are not competing alternatives so much as layers. A RAG system still needs a prompt telling the model how to use the retrieved documents. A tool-using agent still needs a prompt describing its goal and the tools. A fine-tuned model is still called with a prompt. Prompt engineering is the layer that is always present.

Lever	Changes	Best for	Iteration speed	Main cost
Prompting	Model input	Behavior, format, task framing	Minutes	Tokens per call
Fine-tuning	Model weights	Style, narrow tasks at scale	Days to weeks	Data and training
RAG	Context content	Large or changing knowledge	Hours to days	Search infrastructure
Tool use	What model can do	Live data, actions	Hours to days	Integration, safety

A rule of thumb for choosing

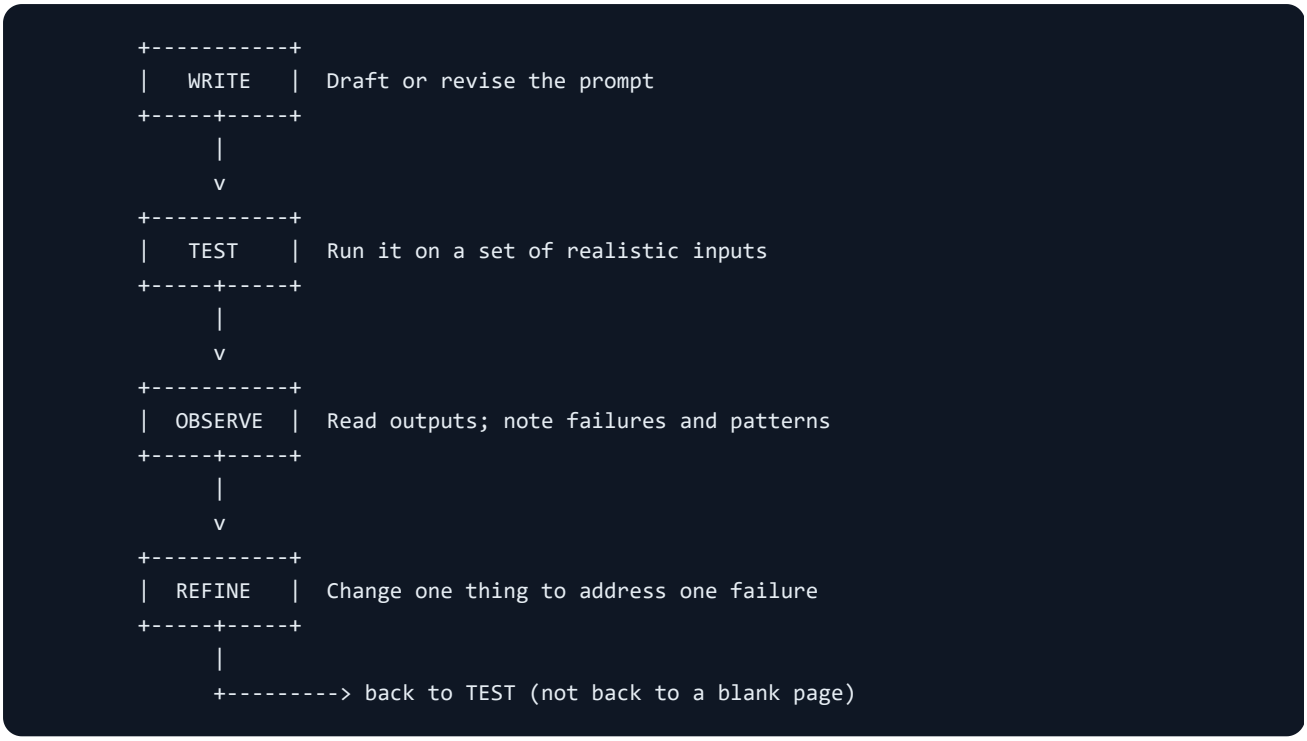
Start with prompting. It is the cheapest experiment, and it tells you what the model can do with the right instructions. If the model fails because it lacks *knowledge* — it does not know your products, your policies, or today's inventory — reach for retrieval or tools. If it fails because it cannot hold a *behavior* consistently even with a well-crafted prompt and examples, or because your prompt has grown so long that cost and latency hurt, consider fine-tuning. If it fails because the task genuinely requires *doing* something — checking an order status, issuing a refund — you need tools.

A frequent and costly mistake is jumping to fine-tuning to fix a problem that a clearer prompt would have solved in an afternoon. The opposite mistake also happens: stuffing an entire product catalog into every prompt because "prompting is simpler," when retrieval would be cheaper, faster, and more accurate.

> **TIP**
> Before proposing fine-tuning, write down the best prompt you can, with examples, and measure it on a test set (Chapter 23). That prompt becomes your baseline — and very often, it turns out to be good enough.

The Iterative Loop

Nobody writes a production-quality prompt on the first attempt. Not experienced practitioners, not the people who train the models. What distinguishes experienced prompt engineers is not their first draft but their loop: a disciplined cycle of writing, testing, observing, and refining.



Write

Start with the clearest statement of the task you can manage. Include the goal, the audience, the essential context, and the output format. Do not try to anticipate every edge case yet; you do not know which edge cases matter until you test.

Test

Run the prompt against a set of inputs, not a single one. Even five to ten varied examples reveal far more than one. Include easy cases, typical cases, and a few awkward ones: an empty input, a very long input, an input in another language, an input that tries to change the model's instructions. As your project matures, this informal set becomes a proper evaluation suite (Chapter 23).

Observe

Read the outputs. Actually read them, rather than skimming for a feeling of success. Look for patterns in the failures. Is the model consistently too long? Does it invent product features? Does it ignore the format on inputs with unusual structure? Write down what you see, because "it feels a bit off" is not something you can fix.

Refine

Change the prompt to address a specific observed failure, and ideally change one thing at a time so you can tell what helped. Then test again on the *whole* set, not just the failing case. A fix for one input can break three others; this is the most common way prompts regress.

> **WARNING**
>
> The single most common prompt-engineering failure in production is "fixed the complaint, broke everything else." Someone reports a bad output, an engineer adds a sentence to the prompt, verifies that the reported case now works, and ships. Without re-running a broader test set, nobody notices that the new sentence changed behavior elsewhere.

The loop in practice

Here is a compressed example of the loop for one Orbit Store task: classifying support tickets.

Iteration 1 — prompt

```
Classify this support ticket into a category.

Ticket: {ticket_text}
```

Testing on ten tickets shows the model inventing categories — "Shipping Concern," "Delivery Issue," "Late Package" — for what should be the same bucket. Observation: the model does not know the allowed categories.

Iteration 2 — prompt

```
Classify this support ticket into exactly one of these categories:
shipping, returns, payments, account, product_question, other

Ticket: {ticket_text}
```

Categories are now consistent, but some outputs are full sentences ("This ticket is about shipping because..."), which breaks the router that reads the result. Observation: the output format is unspecified.

Iteration 3 — prompt

```
Classify the support ticket below into exactly one of these categories:
shipping, returns, payments, account, product_question, other

Guidance:
- A refund request for an item already returned is "payments".
- A request to start a return is "returns".
- Use "other" only if no category fits.

Respond with the category name only, in lowercase, with no other text.

Ticket: {ticket_text}
```

Each iteration fixed a problem that was *observed*, not imagined. The guidance about refunds versus returns was added because tests showed the model splitting those cases inconsistently. That is the loop working as intended. Chapter 14 takes this classifier much further.

Common Myths

Prompt engineering has accumulated a folklore of tips, templates, and "one weird trick" posts. Some of that folklore was once true for specific models; much of it was never true at all. Here are the myths that cause the most wasted effort.

Myth 1: There are magic words

The idea that certain phrases unlock hidden capability — "You are a world-class expert," "Take a deep breath," promises of tips, threats of consequences — is persistent and mostly misleading. Some phrasings have produced measurable effects on specific models in specific experiments. But those effects tend to be small, inconsistent across models, and prone to disappearing when the model is updated. Building a production system on them is building on sand.

What reliably helps is *information*: telling the model what the task is, who it is for, what good output looks like, and why the constraints exist. A role statement can be useful when it conveys real information about perspective, vocabulary, and priorities (Chapter 8 covers this), but its value comes from that information, not from flattery.

Prompt — before

```
You are the world's greatest customer service genius with 50 years of
experience. This is extremely important. Answer this customer perfectly.

Customer: My order hasn't arrived.
```

Prompt — after

```
You are a support assistant for Orbit Store, an online marketplace.
Customers contact you about orders placed with independent merchants.

When a customer reports a missing order:
- Ask for the order number if they have not given it.
- Explain that standard delivery takes 3-7 business days from dispatch.
- If it has been more than 7 business days, offer to open a delivery
  investigation with the merchant.
Keep replies under 120 words and do not promise refunds; refunds are
decided by the merchant.

Customer: My order hasn't arrived.
```

The "before" prompt spends its words on emphasis and gives the model nothing to act on. The "after" prompt contains no superlatives at all, yet it will produce far better answers because it tells the model what Orbit Store's actual process is.

Myth 2: Just ask nicely

A related myth holds that politeness — "please," "thank you," "if you don't mind" — materially improves output. Polite phrasing is harmless and can make prompts pleasant for humans to maintain, and it may nudge tone in customer-facing text. But it is not a substitute for clarity. "Please write a good summary" is still an underspecified request. We write prompts in a courteous, direct register in this book simply because that is clear writing, not because courtesy is a technique.

Myth 3: Longer is always better

Once people discover that adding context helps, they sometimes overcorrect and treat prompt length as a proxy for quality. Long prompts have real costs. They consume context window space and money on every call. They increase latency. And they introduce opportunities for contradiction: the instruction on line 12 and the instruction on line 87 may disagree, and the model has to guess which wins.

The right length is the length that conveys what the model needs and nothing it does not. A rule that addresses a failure you have actually observed earns its place. A paragraph of generic advice ("be accurate, be helpful, be thorough, think carefully") usually does not, because a well-trained model already tries to do those things.

Prompt — before

```
You are a helpful, accurate, thorough, and careful assistant. Always be
helpful. Always be accurate. Never make mistakes. Think carefully about
everything. Be thorough but also concise. Make sure your answer is high
quality. Double-check your work. Be professional. Be friendly. Summarize
the review below.

Review: {review_text}
```

Prompt — after

```
Summarize the product review below for a merchant dashboard.
Write one sentence (max 25 words) stating the customer's main point,
then list any specific product defects mentioned, or "none".

Review: {review_text}
```

The "after" prompt is shorter and more effective, because every sentence specifies something the model could not otherwise know: the audience, the length, and the exact structure.

Myth 4: One perfect prompt exists

There is no single optimal prompt waiting to be discovered. Many different prompts perform equally well on a given task, and the best one for one model may be mediocre for another. Aim for a prompt that is clear, tested, and good enough for your quality bar, then move on and keep monitoring it.

Myth 5: A prompt that works on one model works on all of them

Models from different providers — and different versions from the same provider — have different training, different defaults, and different sensitivities. A prompt tuned for one model may be too verbose for another, may rely on a formatting convention the other ignores, or may trigger different refusal behavior. Clear, well-structured prompts transfer much better than trick-laden ones, but you should always re-test when you switch models.

Three Settings: Chat, Applications, and Agents

The same principles apply wherever you use a language model, but the practical emphasis shifts depending on the setting. It helps to distinguish three.

Chat interfaces

In a chat interface, you are both the prompt engineer and the user. You can see every response and correct course immediately. Prompts are usually informal and conversational, and iteration happens across turns: "shorter," "now make it more formal," "you missed the second point." The stakes of a single weak prompt are low because you are right there to fix it.

Good chat prompting still pays off. Front-loading context ("I'm preparing a presentation for non-technical executives; here is the data...") saves several rounds of correction. Many chat products also let you set persistent instructions or project-level context, which behave much like a system prompt.

API-driven applications

In an application, your prompt is a template executed by code, often thousands of times, on inputs you never see. No human is standing by to rephrase. The prompt must handle the full distribution of real inputs, produce output that downstream code can parse, and fail gracefully when an input is strange. Here prompts are versioned like code, tested against evaluation sets, and changed carefully. Most of this book is written with this setting in mind, because it is where prompt quality has the most leverage.

Agents

An agent is a system in which the model decides what to do next over multiple steps, typically by calling tools and reading their results until a goal is met. The prompt now has to describe not just a task but a way of working: the goal, the available tools and when to use them, when to stop, when to ask a human, and what must never be done without confirmation. The context changes at every step as tool results accumulate. Prompt engineering for agents blends instruction writing with system design, and we devote Part IV of the book to it.

Aspect	Chat	Application	Agent
Who sees output	You, immediately	Code or end users	Tools, then users
Iteration	Turn by turn	Test set, then deploy	Test runs, traces

Main risk	Wasted time	Silent bad outputs	Wrong actions
Prompt style	Conversational	Templated, strict	Goals, tools, limits
Key chapters	4, 8, 15	9, 14, 23	20, 21, 22

The same request looks different across the three settings. Consider asking for help with a product description.

Chat — a person typing

```
I sell handmade ceramic mugs on Orbit Store. Can you help me write a
short, friendly description for a 350 ml speckled blue mug? It's
dishwasher safe but not microwave safe.
```

Application — a template filled by code

```
Write a product description for an Orbit Store listing.

Rules:
- 60-90 words, one paragraph.
- Use only the facts in the listing data. Do not add materials,
  dimensions, or care instructions that are not listed.
- If a care instruction says "not microwave safe", include it.

Listing data:
{listing_json}

Return only the description text.
```

Agent — part of an operations agent's instructions

```
Your goal is to make sure every new listing in the merchant's catalog
has a description that meets Orbit Store's content rules.

Available tools: list_new_listings, get_listing, draft_description,
update_listing.

Work through new listings one at a time. For each listing, draft a
description, check it against the content rules, and call
update_listing only if the draft passes. If a listing is missing
required facts (material, size, or care), do not guess: add it to a
list of items that need merchant input and report that list at the end.
```

Meet Orbit Store

Abstract advice is easy to agree with and hard to apply. To keep this book concrete, we follow a single fictional company through every chapter. Orbit Store is an online marketplace where thousands of independent merchants sell everything from kitchenware to bicycle parts. Orbit Store handles the

storefront, payments, and customer support; merchants handle inventory and shipping. The company has decided to use LLMs across its business, and each use case brings its own prompting challenges.

The customer-support assistant

Customers message Orbit Store about late deliveries, returns, payment problems, and product questions. A support assistant answers common questions directly and hands off to human agents when needed. The challenges: it must follow policy exactly (it cannot promise refunds it is not authorized to give), stay friendly under pressure, avoid inventing order details, and resist customers who try to talk it into ignoring its rules. This assistant is the subject of the Chapter 27 case study.

The product-description writer

Merchants upload listings with sparse data — a title, a few attributes, sometimes a sentence or two. The description writer turns that data into polished, on-brand copy. The challenges: consistent voice across thousands of listings, strict adherence to supplied facts (an invented "waterproof" claim can cause returns and legal exposure), and length and format rules imposed by the storefront.

The review summarizer

Popular products accumulate hundreds of reviews. The summarizer produces short "what customers say" blurbs for shoppers and defect reports for merchants. The challenges: faithfully representing mixed opinions, not overweighting a single vivid review, and separating product problems from shipping complaints.

The support-ticket classifier and router

Every incoming ticket is categorized and routed to the right queue, with a priority. The challenges: a fixed label set, consistent handling of ambiguous tickets, machine-readable output, and very high volume, which makes cost and latency important.

Catalog data extraction

Suppliers send product specification sheets as PDFs with wildly different layouts. An extraction pipeline pulls structured fields — dimensions, materials, certifications, prices — into Orbit Store's catalog. The challenges: strict schemas, handling missing or conflicting values honestly rather than guessing, and long, messy documents.

The SQL analytics assistant

Merchants want answers such as "Which products had the highest return rate last quarter?" without writing SQL. The analytics assistant translates questions into queries against a read-only view of the merchant's data. The challenges: correctness against a real schema, safety (no data from other merchants, no destructive queries), and explaining results in plain language.

The shopping agent

A conversational shopping agent helps customers find products, compare options, and build a cart, using tools to search the catalog, check stock, and look up shipping estimates. The challenges: choosing the right tool at the right time, not hallucinating products or prices, and never completing a purchase without explicit customer confirmation.

The merchant-operations agent

Behind the scenes, an operations agent helps merchants keep their catalogs healthy: flagging listings with missing data, drafting fixes, reconciling inventory reports, and preparing summaries. The challenges: long multi-step workflows, safe use of write actions, clear reporting, and knowing when to stop and ask a human. This agent is the focus of the Chapter 28 case study.

Use case	Main techniques	Chapters
Support assistant	System prompts, tone, guardrails	3, 8, 24, 27
Description writer	Clarity, examples, style	4, 5, 15
Review summarizer	Summarization, grounding	10, 13
Ticket classifier	Classification, JSON	9, 14
Catalog extraction	Extraction, schemas	9, 10, 13
SQL assistant	Text-to-SQL, validation	16, 17
Shopping agent	Tools, RAG, agents	19, 20, 21
Operations agent	Planning, context	21, 22, 28

These use cases were chosen because together they cover nearly every pattern you will meet in practice: free-form generation, strict structured output, grounded answers from documents, classification at scale, code generation, multi-step tool use, and the security concerns that come with exposing a model to untrusted input. When you finish the book, you should be able to map almost any LLM project you encounter onto one or more of them.

> **NOTE**
>
> Orbit Store is fictional, and its policies (delivery windows, return periods, and so on) are invented for illustration. When you adapt an example, replace every policy detail with your own organization's real rules. A prompt that confidently states the wrong policy is worse than no prompt at all.

What Prompt Engineering Is Not

Before we outline the rest of the book, it is worth drawing a few boundaries, because many disappointments come from expecting prompts to do things they cannot.