AI & LLMS

Prompt Engineering for Developers

Reliable patterns, structured outputs and tool use

Houssam Kodad

PDF · DATAFORGE BOOKS

Questions about this sample or the full edition: **support@dataforgebooks.com**

CONTENTS

Table of Contents

CHAPTER 01

Prompting as Software Design

Prompt engineering has a reputation for being a bag of tricks — magic words that coax better answers out of a model. The reality, for anyone building software rather than chatting, is more disciplined and more durable: a prompt is an interface, and the patterns that make it reliable look a great deal like good software design. This short, example-driven book teaches those patterns.

This chapter sets the tone for everything that follows. We argue that prompts are interfaces to be designed rather than incantations to be discovered, confront the variance that makes them harder to work with than ordinary functions, and make the case for treating prompts as versioned, tested code. These are the mindsets that separate a prompt you can build a product on from one that works by luck.

The book is aimed squarely at developers, so the examples are concrete and the techniques are ones you can apply inside a real application today. By the end you will write prompts that produce structured, parseable output, decompose hard tasks into reliable steps, and use tools — and, crucially, prompts you can test and improve systematically rather than tweak by superstition.

Why Prompts Are Interfaces

When you call a model, you are calling a function whose entire behaviour is defined by the text you send. The prompt is its interface: it specifies the task, the constraints, the format of the answer, and the context the model should use. Treating that text casually is like writing an API with no contract and hoping callers guess your intentions correctly — it works in a demo and fails the moment reality varies.

Good prompts, like good interfaces, are explicit. They state what is wanted, what to avoid, and exactly what shape the output must take. The vague, conversational prompt that works fine in a chat window is precisely the wrong model for software, where the same prompt will run unattended thousands of times against inputs you never saw. The whole book is, in a sense, about applying interface-design discipline to that text.

Determinism and Variance

Unlike an ordinary function, a model can return different outputs for the same input, and this variance is the central difficulty of building on language models. A prompt that works nine times out of ten still fails one user in ten, and — worse for a developer — you cannot reliably reproduce that failure to

debug it. Accepting and managing this is the first real skill of the craft.

Managing variance is a recurring theme rather than a single fix. Lowering temperature makes output more consistent. Constraining the response to a strict format shrinks the space in which things can go wrong. Asking the model to follow an explicit structure, and validating what comes back, turns silent failures into detectable ones. You will never make a model perfectly deterministic, but you can make its useful behaviour overwhelmingly likely — and verifiable.

```
# Constrain output so failures are detectable, not silent.
prompt = '''Classify the support ticket. Respond with JSON only:
{"category": "billing|technical|account", "urgent": true|false}

Ticket: ''' + ticket_text

raw = model.complete(prompt, temperature=0)   # low temperature = consistent
result = json.loads(raw)                       # validate; never trust blindly
```

Anatomy of a Reliable Prompt

A production prompt has parts, each doing a job. Clear instructions state the task and its constraints. Delimiters separate the instructions from the data so the model cannot confuse the two — a subtle but important defence against both errors and injection. Examples, where used, show the model the exact shape of a good answer. And a role or system message sets the overall behaviour and tone.

Assembling these deliberately, rather than writing one run-on paragraph and hoping, is what makes a prompt robust across the range of inputs production will throw at it. We break down each component with examples, because the difference between a prompt that works on your three test cases and one that works on ten thousand real ones is almost always this structure — not a cleverer turn of phrase.

Structured Outputs

For software, the most valuable thing a model can do is return data your code can act on. Free-form prose is hard to build on; a strict JSON object that conforms to a known schema is a reliable component you can wire into the rest of your system. Asking for structured output, specifying the schema precisely, and validating the result is the technique that unlocks most real applications.

Validation is not optional, because even a well-prompted model occasionally returns malformed output, and a production system must handle that gracefully rather than crash. The robust pattern is to request the structure, parse it, validate against the schema, and have a clear fallback or retry when validation fails. We treat this loop — request, parse, validate, recover — as the backbone of building

on language models.

Decomposition and Chaining

Asking a model to do too much in one prompt is a common cause of unreliable output. A complex task — read this document, extract the key facts, judge them against a policy, and draft a response — is more reliable when broken into a chain of focused steps, each with its own simple prompt, each easy to test and reason about in isolation.

Decomposition is the same instinct that leads good programmers to write small functions rather than one giant one, and it pays off the same way: each step is simpler, more testable, and more reusable. Chaining prompts together, and routing between different prompts based on intermediate results, lets you build sophisticated behaviour out of pieces you can actually trust. We develop this pattern throughout the book.

Reasoning and Tool Use

For tasks that require working through steps — arithmetic, logic, multi-step analysis — asking the model to reason explicitly before answering often improves accuracy markedly, because it gives the model room to work rather than forcing an immediate guess. Related patterns have the model check its own answer or generate several and reconcile them, trading a little extra cost for meaningfully better reliability.

Tool use takes this further by letting the model call functions you define — a search, a calculation, a database lookup — to fetch facts or perform actions it cannot do reliably on its own. Describing tools clearly and validating their arguments keeps your code in control of what is actually allowed. Together, reasoning patterns and tool use are what let a prompt-driven application do dependable work rather than merely produce plausible text.

Treating Prompts as Code

If a prompt is an interface that ships in your product, it deserves what the rest of your code gets: version control, review, and tests. A prompt edited live in a console with no history is a liability — when quality drops, you cannot tell what changed or roll back to what worked. Bringing prompts into the repository, and changing them through review, is the first step to engineering rather than tinkering.

The other half is evaluation. Maintain a small set of representative inputs with expected outputs, and run every prompt change against it before shipping, so improvement and regression are measured

rather than felt. This discipline, developed across the final chapters, is what turns prompt engineering from a dark art into ordinary, dependable software work — which is exactly what building a real product on language models requires.

This is a free sample

You've reached the end of the sample chapter.

Get the complete book — every chapter, fully worked — at **dataforgebooks.com**.

FULL EDITION · 144 PAGES · PDF

Read the full title at **dataforgebooks.com**

Questions? support@dataforgebooks.com