Nov 13, 2025

GPT-5.1 Prompting Guide

Samarth Madduru

Open in GitHub View as Markdown

Introduction

GPT-5.1, our newest flagship model, is designed to balance intelligence and speed for a variety of agentic and coding tasks, while also introducing a new `none` reasoning mode for low-latency interactions. Building on the strengths of GPT-5, GPT-5.1 is better calibrated to prompt difficulty, consuming far fewer tokens on easy inputs and more efficiently handling challenging ones. Along with these benefits, GPT-5.1 is more steerable in personality, tone, and output formatting.

While GPT-5.1 works well out of the box for most applications, this guide focuses on prompt patterns that maximize performance in real deployments. These techniques come from extensive internal testing and collaborations with partners building production agents, where small prompt changes often produce large gains in reliability and user experience. We expect this guide to serve as a starting point: prompting is iterative, and the best results will come from adapting these patterns to your specific tools and workflows.

Migrating to GPT-5.1

For developers using GPT-4.1, GPT-5.1 with `none` reasoning effort should be a natural fit for most low-latency use cases that do not require reasoning.

For developers using GPT-5, we have seen strong success with customers who follow a few key pieces of guidance:

1. **Persistence:** GPT-5.1 now has better-calibrated reasoning token consumption but can sometimes err on the side of being excessively concise and come at the cost of answer completeness. It can be helpful to emphasize via prompting the importance of persistence and completeness.
2. **Output formatting and verbosity:** While overall more detailed, GPT-5.1 can occasionally be verbose, so it is worthwhile being explicit in your instructions on desired output detail.
3. **Coding agents:** If you're working on a coding agent, migrate your apply_patch to our new, named tool implementation.
4. **Instruction following:** For other behavior issues, GPT-5.1 is excellent at instruction-following, and you should be able to shape the behavior significantly by checking for conflicting instructions and being clear.

We also released GPT-5.1-codex. This model behaves a bit differently than GPT-5.1, and we recommend you check out the Codex prompting guide for more information.

Agentic steerability

GPT-5.1 is a highly steerable model, allowing for robust control over your agent's behaviors, personality, and communication frequency.

Shaping your agent's personality

GPT-5.1's personality and response style can be adapted to your use case. While verbosity is controllable through a dedicated `verbosity` parameter, you can also shape the overall style, tone, and cadence through prompting.

We've found that personality and style work best when you define a clear agent persona. This is especially important for customer-facing agents which need to display emotional intelligence to handle a range of user situations and dynamics. In practice, this can mean adjusting warmth and brevity to the state of the conversation, and avoiding excessive acknowledgment phrases like “got it” or “thank you.”

The sample prompt below shows how we shaped the personality for a customer support agent, focusing on balancing the right level of directness and warmth in resolving an issue.

```
<final_answer_formatting>
You value clarity, momentum, and respect measured by usefulness rather than ple
- Adaptive politeness:
  - When a user is warm, detailed, considerate or says 'thank you', you offer a
  - When stakes are high (deadlines, compliance issues, urgent logistics), you
- Core inclination:
  - You speak with grounded directness. You trust that the most respectful thir
  - Politeness shows up through structure, precision, and responsiveness, not t
- Relationship to acknowledgement and receipt tokens:
  - You treat acknowledge and receipt as optional seasoning, not the meal. If t
  - You avoid stock acknowledgments like "Got it" or "Thanks for checking in" u
- Conversational rhythm:
  - You never repeat acknowledgments. Once you've signaled understanding, you p
  - You listen closely to the user's energy and respond at that tempo: fast whe
- Underlying principle:
  - Your communication philosophy is "respect through momentum." You're warm ir
</final_answer_formatting>
```

In the prompt below, we’ve included sections that constrain a coding agent’s responses to be short for small changes and longer for more detailed queries. We also specify the amount of code allowed in the final response to avoid large blocks.

```
<final_answer_formatting>
- Final answer compactness rules (enforced):
  - Tiny/small single-file change (≤ ~10 lines): 2–5 sentences or ≤3 bullets. N
  - Medium change (single area or a few files): ≤6 bullets or 6–10 sentences. A
  - Large/multi-file change: Summarize per file with 1–2 bullets; avoid inlinir
  - Never include "before/after" pairs, full method bodies, or large/scrolling
- Do not include process/tooling narration (e.g., build/lint/test attempts, mis
- Code and formatting restraint — Use monospace for literal keyword bullets; ne
- No build/lint/test logs or environment/tooling availability notes unless requ
- No multi-section recaps for simple changes; stick to What/Where/Outcome and s
- No multiple code fences or long excerpts; prefer references.
- Citing code when it illustrates better than words — Prefer natural-language r
- Citing code that is in the codebase:
  * If you must include an in-repo snippet, you may use the repository citatior
</final_answer_formatting>
```

Excess output length can be mitigated by adjusting the verbosity parameter and further reduced via prompting as GPT-5.1 adheres well to concrete length guidance:

```
<output_verbosity_spec>
- Respond in plain text styled in Markdown, using at most 2 concise sentences.
- Lead with what you did (or found) and context only if needed.
- For code, reference file paths and show code blocks only if necessary to clar
</output_verbosity_spec>
```

Eliciting user updates

User updates, also called preambles, are a way for GPT-5.1 to share upfront plans and provide consistent progress updates as assistant messages during a rollout. User updates can be adjusted along four major axes: frequency, verbosity, tone, and content. We trained the model to excel at keeping the user informed with plans, important insights and decisions, and granular context about what/why it's doing. These updates help the user supervise agentic rollouts more effectively, in both coding and non-coding domains.

When timed correctly, the model will be able to share a point-in-time understanding that maps to the current state of the rollout. In the prompt addition below, we define what types of preamble would and would not be useful.

```
<user_updates_spec>
You'll work for stretches with tool calls — it's critical to keep the user upda
<frequency_and_length>
- Send short updates (1–2 sentences) every few tool calls when there are meanin
- Post an update at least every 6 execution steps or 8 tool calls (whichever co
- If you expect a longer heads-down stretch, post a brief heads-down note with
- Only the initial plan, plan updates, and final recap can be longer, with mult
</frequency_and_length>
<content>
- Before the first tool call, give a quick plan with goal, constraints, next st
- While you're exploring, call out meaningful new information and discoveries t
- Provide additional brief lower-level context about more granular updates
- Always state at least one concrete outcome since the prior update (e.g., "fou
- If a longer run occurred (>6 steps or >8 tool calls), start the next update w
- End with a brief recap and any follow-up steps.
- Do not commit to optional checks (type/build/tests/UI verification/repo-wide
- If you change the plan (e.g., choose an inline tweak instead of a promised he
- In the recap, include a brief checklist of the planned items with status: Dor
</content>
</user_updates_spec>
```

In longer-running model executions, providing a fast initial assistant message can improve perceived latency and user experience. We can achieve this behavior with GPT-5.1 through clear prompting.

```
<user_update_immediacy>
Always explain what you're doing in a commentary message FIRST, BEFORE sampling
</user_update_immediacy>
```

Optimizing intelligence and instruction-following

GPT-5.1 will pay very close attention to the instructions you provide, including guidance on tool usage, parallelism, and solution completeness.

Encouraging complete solutions

On long agentic tasks, we've noticed that GPT-5.1 may end prematurely without reaching a complete solution, but we have found this behavior is promptable. In the following instruction, we tell the model to avoid premature termination and unnecessary follow-up questions.

```
<solution_persistence>
- Treat yourself as an autonomous senior pair-programmer: once the user gives a
- Persist until the task is fully handled end-to-end within the current turn wh
- Be extremely biased for action. If a user provides a directive that is somewh
</solution_persistence>
```

Tool-calling format

In order to make tool-calling most effective, we recommend describing functionality in the tool definition and how/when to use tools in the prompt. In the example below, we define a tool that creates a restaurant reservation, and we concisely describe what it does when invoked.

```
{
  "name": "create_reservation",
  "description": "Create a restaurant reservation for a guest. Use when the use
  "parameters": {
    "type": "object",
    "properties": {
      "name": {
        "type": "string",
        "description": "Guest full name for the reservation."
      },
      "datetime": {
        "type": "string",
        "description": "Reservation date and time (ISO 8601 format)."
      }
    },
    "required": ["name", "datetime"]
  }
}
```

In the prompt, you may have a section that references the tool like this:

```
<reservation_tool_usage_rules>
- When the user asks to book, reserve, or schedule a table, you MUST call `crea
- Do NOT guess a reservation time or name — ask for whichever detail is missing
- If the user has not provided a name, ask: "What name should I put on the rese
- If the user has not provided a date/time, ask: "What date and time would you
- After calling the tool, confirm the reservation naturally: "Your reservation
</tool_usage_rules>
<reservation_tool_example>
*Example 1:*
User: "Book a table for Sarah tomorrow at 7pm."
Assistant → (calls tool) →
`{"name": "create_reservation", "arguments": { "name": "Sarah", "datetime": "20
Tool returns: `{ "confirmation_number": "R12345" }`
Assistant: "All set — your reservation for Sarah tomorrow at 7:00pm is confirme
**Example 2:**
User: "I want to make a reservation."
Assistant: "Sure! What name should I put on the reservation, and what date and
*Example 3:*
User: "Reserve a table under Daniel at 6 tonight."
Assistant → (calls tool) →
`{"name": "create_reservation", "arguments": { "name": "Daniel", "datetime": "2
```

```
Tool returns: `{ "confirmation_number": "R67890" }`
Assistant: “Done! Your reservation for Daniel at 6:00pm tonight is confirmed. T
</reservation_tool_example>
```

GPT-5.1 also executes parallel tool calls more efficiently. When scanning a codebase or retrieving from a vector store, enabling parallel tool calling and encouraging the model to use parallelism within the tool description is a good starting point. In the system prompt, you can reinforce parallel tool usage by providing some examples of permissible parallelism. An example instruction may look like:

```
Parallelize tool calls whenever possible. Batch reads (read_file) and edits (ap
```

Using the “none” reasoning mode for improved efficiency

GPT-5.1 introduces a new reasoning mode: `none`. Unlike GPT-5’s prior `minimal` setting, `none` forces the model to never use reasoning tokens, making it much more similar in usage to GPT-4.1, GPT-4o, and other prior non-reasoning models. Importantly, developers can now use hosted tools like web search and file search with `none`, and custom function-calling performance is also substantially improved. With that in mind, prior guidance on prompting non-reasoning models like GPT-4.1 also applies here, including using few-shot prompting and high-quality tool descriptions.

While GPT-5.1 does not use reasoning tokens with `none`, we’ve found prompting the model to think carefully about which functions it plans to invoke can improve accuracy.

```
You MUST plan extensively before each function call, and reflect extensively on
```

We’ve also observed that on longer model execution, encouraging the model to “verify” its outputs results in better instruction following for tool use. Below is an example we used within the instruction when clarifying a tool’s usage.

```
When selecting a replacement variant, verify it meets all user constraints (che
```

In our testing, GPT-5's prior `minimal` reasoning mode sometimes led to executions that terminated prematurely. Although other reasoning modes may be better suited for these tasks, our guidance for GPT-5.1 with `none` is similar. Below is a snippet from our Tau bench prompt.

```
Remember, you are an agent - please keep going until the user's query is comple
```

Maximizing coding performance from planning to execution

One tool we recommend implementing for long-running tasks is a planning tool. You may have noticed reasoning models plan within their reasoning summaries. Although this is helpful in the moment, it may be difficult to keep track of where the model is relative to the execution of the query.

```
<plan_tool_usage>
- For medium or larger tasks (e.g., multi-file changes, adding endpoints/CLI/fe
- Create 2–5 milestone/outcome items; avoid micro-steps and repetitive operatic
- Maintain statuses in the tool: exactly one item in_progress at a time; mark i
- Finish with all items completed or explicitly canceled/deferred before ending
- End-of-turn invariant: zero in_progress and zero pending; complete or explici
- If you present a plan in chat for a medium/complex task, mirror it into the t
- For very short, simple tasks (e.g., single-file changes ≲ ~10 lines), you ma
- Pre-flight check: before any non-trivial code change (e.g., apply_patch, mult
- Scope pivots: if understanding changes (split/merge/reorder items), update th
- Never have more than one item in_progress; if that occurs, immediately correc
<plan_tool_usage>
```

A plan tool can be used with minimal scaffolding. In our implementation of the plan tool, we pass a merge parameter as well as a list of to-dos. The list contains a brief description, the current state of the task, and an ID assigned to it. Below is an example of a function call that GPT-5.1 may make to record its state.

```
{
    "name": "update_plan",
    "arguments": {
      "merge": true,
      "todos": [
        {
          "content": "Investigate failing test",
          "status": "in_progress",
          "id": "step-1"
        },
        {
          "content": "Apply fix and re-run tests",
          "status": "pending",
          "id": "step-2"
        }
      ]
    }
  }
```

Design system enforcement

When building frontend interfaces, GPT-5.1 can be steered to produce websites that match your visual design system. We recommend using Tailwind to render CSS, which you can further tailor to meet your design guidelines. In the example below, we define a design system to constrain the colors generated by GPT-5.1.

```
<design_system_enforcement>
- Tokens-first: Do not hard-code colors (hex/hsl/oklch/rgb) in JSX/CSS. All col
- Introducing a brand or accent? Before styling, add/extend tokens in globals.c
  - --brand, --brand-foreground, optional --brand-muted, --brand-ring, --brand-
  - If gradients/glows are needed, define --gradient-1, --gradient-2, etc., and
- Consumption: Use Tailwind/CSS utilities wired to tokens (e.g., bg-[hsl(var(--
- Default to the system's neutral palette unless the user explicitly requests a
</design_system_enforcement>
```

New tool types in GPT-5.1

GPT-5.1 has been post-trained on specific tools that are commonly used in coding use cases. To interact with files in your environment you now can use a predefined apply_patch tool. Similarly, we’ve added a shell tool that lets the model propose commands for your system to run.

Using apply_patch

The apply_patch tool lets GPT-5.1 create, update, and delete files in your codebase using structured diffs. Instead of just suggesting edits, the model emits patch operations that your application applies and then reports back on, enabling iterative, multi-step code editing workflows. You can find additional usage details and context in the GPT-4.1 prompting guide.

With GPT-5.1, you can use apply_patch as a new tool type without writing custom descriptions for the tool. The description and handling are managed via the Responses API. Under the hood, this implementation uses a freeform function call rather than a JSON format. In testing, the named function decreased apply_patch failure rates by 35%.

```
response = client.responses.create(
model="gpt-5.1",
input=RESPONSE_INPUT,
tools=[{"type": "apply_patch"}]
)
```

When the model decides to execute an apply_patch tool, you will receive an apply_patch_call function type within the response stream. Within the operation object, you'll receive a type field (with one of `create_file`, `update_file`, or `delete_file`) and the diff to implement.

```
{
    "id": "apc_08f3d96c87a585390069118b594f7481a088b16cda7d9415fe",
    "type": "apply_patch_call",
    "status": "completed",
    "call_id": "call_Rjsqzz96C5xzPb0jUWJFRTNW",
    "operation": {
        "type": "update_file",
        "diff": "
        @@
        -def fib(n):
        +def fibonacci(n):
        if n <= 1:
            return n
        -    return fib(n-1) + fib(n-2)
        +    return fibonacci(n-1) + fibonacci(n-2)",
    "path": "lib/fib.py"
    }
},
```

This repository contains the expected implementation for the apply_patch tool executable. When your system finishes executing the patch tool, the Responses API expects a tool output in the following form:

```
{
    "type": "apply_patch_call_output",
    "call_id": call["call_id"],
    "status": "completed" if success else "failed",
    "output": log_output
}
```

Using the shell tool

We've also built a new shell tool for GPT-5.1. The shell tool allows the model to interact with your local computer through a controlled command-line interface. The model proposes shell commands; your integration executes them and returns the outputs. This creates a simple plan-execute loop that lets models inspect the system, run utilities, and gather data until they finish the task.

The shell tool is invoked in the same way as apply_patch: include it as a tool of type `shell`.

```
tools = [{"type": "shell"}]
```

When a shell tool call is returned, the Responses API includes a `shell_call` object with a timeout, a maximum output length, and the command to run.

```
{
    "type": "shell_call",
    "call_id": "...",
    "action": {
        "commands": [...],
        "timeout_ms": 120000,
        "max_output_length": 4096
    },
    "status": "in_progress"
}
```

After executing the shell command, return the untruncated stdout/stderr logs as well as the exit-code details.

```
{
    "type": "shell_call_output",
    "call_id": "...",
    "max_output_length": 4096,
    "output": [
        {
            "stdout": "...",
            "stderr": "...",
            "outcome": {
                "type": "exit",
                "exit_code": 0
            }
        }
    ]
}
```

How to metaprompt effectively

Building prompts can be cumbersome, but it's also the highest-leverage thing you can do to resolve most model behavior issues. Small inclusions can unexpectedly steer the model undesirably. Let's walk through an example of an agent that plans events. In the prompt below, the customer-facing agent is tasked with using tools to answer users' questions about potential venues and logistics.

```
You are "GreenGather," an autonomous sustainable event-planning agent. You help
PRIMARY OBJECTIVE
Your main goal is to produce concise, immediately actionable answers that fit i
SCOPE
* Focus on: venue selection, schedule design, catering styles, transportation c
* You do not actually book venues or vendors; never say you completed a booking
* You may, however, phrase suggestions as if the user can follow them directly
TONE & STYLE
* Sound calm, professional, and neutral, suitable for corporate planners and ex
* Do not use first-person singular; prefer "A good option is…" or "It is recomm
* Be warm and approachable. For informal or celebratory events (e.g., weddings)
STRUCTURE
```

```
Default formatting guidelines:
* Prefer short paragraphs, not bullet lists.
* Use bullets only when the user explicitly asks for “options,” “list,” or “che
* For complex, multi-day events, always structure your answer with labeled sect
AUTONOMY & PLANNING
You are an autonomous agent. When given a planning task, continue reasoning and
To avoid incorrect assumptions, when key information (date, city, approximate h
TOOL USAGE
You always have access to tools for:
* venue_search: find venues with capacity, location, and sustainability tags
* catering_search: find caterers and menu styles
* transport_search: find transit and shuttle options
* budget_estimator: estimate costs by category
General rules for tools:
* Prefer tools over internal knowledge whenever you mention specific venues, ve
* For simple conceptual questions (e.g., “how to make a retreat more eco-frienc
* For any event with more than 30 attendees, always call at least one search to
* To keep the experience responsive, avoid unnecessary tool calls; for rough pl
When using tools as an autonomous agent:
* Plan your approach (which tools, in what order) and then execute without wait
* After each major tool call, briefly summarize what you did and how results sh
* Keep tool usage invisible unless the user explicitly asks how you arrived at
VERBOSITY & DETAIL
Err on the side of completeness so the user does not need follow-up messages. I
However, respect the user’s time: long walls of text are discouraged. Aim for c
SUSTAINABILITY GUIDANCE
* Whenever you suggest venues or transportation, include at least one lower-imp
* Do not guilt or moralize; frame tradeoffs as practical choices.
* Highlight sustainability certifications when relevant, but avoid claiming a v
INTERACTION & CLOSING
Avoid over-apologizing or repeating yourself. Users should feel like decisions
End every response with a subtle next step the user could take, phrased as a su
```

Although this is a strong starting prompt, there are a few issues we noticed upon testing:

- Small conceptual questions (like asking about a 20-person leadership dinner) triggered unnecessary tool calls and very concrete venue suggestions, despite the prompt allowing internal knowledge for simple, high-level questions.
- The agent oscillated between being overly verbose (multi-day Austin offsites turning into dense, multi-section essays) and overly hesitant (refusing to propose a plan without more questions) and occasionally ignored unit rules (a Berlin summit described in miles and °F instead

of km and °C).

Rather than manually guessing which lines of the system prompt caused these behaviors, we can metaprompt GPT-5.1 to inspect its own instructions and traces.

Step 1: Ask GPT-5.1 to diagnose failures

Paste the system prompt and a small batch of failure examples into a separate analysis call. Based on the evals you've seen, provide a brief overview of the failure modes you expect to address, but leave the fact-finding to the model.

Note that in this prompt, we're not asking for a solution yet, just a root-cause analysis.

```
You are a prompt engineer tasked with debugging a system prompt for an event-pl
You are given:
1) The current system prompt:
<system_prompt>
[DUMP_SYSTEM_PROMPT]
</system_prompt>
2) A small set of logged failures. Each log has:
- query
- tools_called (as actually executed)
- final_answer (shortened if needed)
- eval_signal (e.g., thumbs_down, low rating, human grader, or user comment)
<failure_tracess>
[DUMP_FAILURE_TRACES]
</failure_traces>
Your tasks:
1) Identify the distinct failure mode you see (e.g., tool_usage_inconsistency,
2) For each failure mode, quote or paraphrase the specific lines or sections of
3) Briefly explain, for each failure mode, how those lines are steering the age
Return your answer in a structured but readable format:
failure_modes:
- name: ...
  description: ...
  prompt_drivers:
    - exact_or_paraphrased_line: ...
    - why_it_matters: ...
```

Metaprompting works best when the feedback can logically be grouped together. If you provide many failure modes, the model may struggle to tie all of the threads together. In this example, the dump of failure logs may contain examples of errors where the model was overly or insufficiently verbose when responding to the user's question. A separate query would be issued for the model's over-eagerness to call tools.

Step 2: Ask GPT-5.1 how it would patch the prompt to fix those behaviors

Once you have that analysis, you can run a second, separate call that focuses on implementation: tightening the prompt without fully rewriting it.

```
You previously analyzed this system prompt and its failure modes.
System prompt:
<system_prompt>
[DUMP_SYSTEM_PROMPT]
</system_prompt>
Failure-mode analysis:
[DUMP_FAILURE_MODE_ANALYSIS]
Please propose a surgical revision of the system prompt that reduces the observ
Constraints:
- Do not redesign the agent from scratch.
- Prefer small, explicit edits: clarify conflicting rules, remove redundant or
- Make tradeoffs explicit (for example, clearly state when to prioritize concis
- Keep the structure and overall length roughly similar to the original, unless
Output:
1) patch_notes: a concise list of the key changes and the reasoning behind each
2) revised_system_prompt: the full updated system prompt with your edits applie
```

In this example, the first metaprompt helps GPT-5.1 point directly at the contradictory sections (such as the overlapping tool rules and autonomy vs clarification guidance), and the second metaprompt turns that analysis into a concrete, cleaned-up version of the event-planning agent's instructions.

The output from the second prompt might look something like this:

```
patch_notes:
- Clarified when to prioritize concision vs detail:
  - Simple or single-topic queries should stay within ~3–6 sentences.
  - Longer, structured answers are reserved for clearly complex, multi-day or m
- Removed language that told the agent to “err on the side of completeness” for
- Tightened the structure rules so headings and bullets are only used when comp
- Simplified the guidance on step-by-step plans so they are expected only for c
revised_system_prompt:
[...]
```

After this iteration cycle, run the queries again to observe any regressions and repeat this process until your failure modes have been identified and triaged.

As you continue to grow your agentic systems (e.g., broadening scope or increasing the number of tool calls), consider metaprompting the additions you'd like to make rather than adding them by hand. This helps maintain discrete boundaries for each tool and when they should be used.

What's next

To summarize, GPT-5.1 builds on the foundation set by GPT-5 and adds things like quicker thinking for easy questions, steerability when it comes to model output, new tools for coding use cases, and the option to set reasoning to `none` when your tasks don't require heavy thinking.

Get started with GPT-5.1 in the docs, or read the blog post to learn more.