How to Prompt GPT-4.1

and a whole bunch of other helpful AI info

KEY CHANGES WITH GPT-4.1

GPT-4.1, which is only available via API at time of writing, follows instructions more closely than GPT-4o.

It also has a much longer context length—1M tokens versus the previous 128k tokens!

1,000,000 tokens is approximately 750,000 words.

Or, nearly four full copies of *Jane Eyre*.

I unpacked what this expanded context means (and where it still falls short) in my December 27 2024 newsletter on **aiwithallie.beehiiv.com**—insights worth considering before you weave a super-long-context model into your daily workflow.

BUT, there are several prompt hacks you should know to get the most out of GPT-4.1, so let's dive in.

PROMPT STRUCTURE FOR GPT-4.1

Start with key task and instructions.
Break instructions into subtasks in bullets or numbered list.
Show examples of the behavior you want. Repeat that information in the instructions as well.
Use markdown to format the prompt (XML also works and JSON is not recommended).

```
# Role and Objective
# Instructions
## Sub-categories for more detailed instructions
# Reasoning Steps
# Output Format
# Examples
## Example 1
# Context
# Final instructions and prompt to think step by step
```

Not working? Check for conflicting information. Increase specificity. Improve examples.

REMINDERS FOR USING GPT-4.1

The example prompt shared by OpenAI was *1400 words with 53 reminders to think thoroughly or carefully*.

Some extra things that stand out to me:

- Don't be vague. Be clear and explicit about what you want/need (more than with 4o!)
- Related to that, you may have to rewrite prompts that you used with previous models
- Continue to keep it formatted by section (ex: debugging vs verification)
- Continue to share context; if your context is long, write the core instructions above *and* below the context!
- Continue to share examples (like how to use a tool)
- Repeat important instructions (yes, even 53 times!)
- If GPT-4.1 misses something critical, add callouts in caps (ex: DO NOT); note that OpenAI explicitly says not to start with this, but their example has 10 of them!
- Ask it to think step by step for more logical output ("First, think carefully step by step about what is needed to answer the query. Then, xyz. Then, xyz.")
- If there is conflicting info, GPT-4.1 tends to follow the info toward the end of the prompt
- Consider additional if-then statements for edge cases ("Always include target customer persona. If you don't have any information on persona, ask the user for information you need.")

AI AGENTS: SPECIFIC ADVICE FOR GPT-4.1 AND AGENTS

To enable more “eager” agentic workflows instead of standard chatbots, include these three reminders to the model:

> To prevent giving up early - “You are an agent - please keep going until the user’s query is completely resolved, before ending your turn and yielding back to the user. Only terminate your turn when you are sure that the problem is solved.”

> To use tools and reduce hallucinations - “If you are not sure about file content or codebase structure pertaining to the user’s request, use your tools to read files and gather the relevant information: do NOT guess or make up an answer.” And devs reading this, be sure to use the tools field and not a separate parser.

> To reflect on planning - “You MUST plan extensively before each function call, and reflect extensively on the outcomes of the previous function calls. DO NOT do this entire process by making function calls only, as this can impair your ability to solve the problem and think insightfully.” GPT-4.1 is *not* a reasoning model, but encouraging chain of thought will boost planning.

EXAMPLE PROMPT FOR GPT-4.1

Note: this prompt is 7 pages long and more easily copied from their Github or Cookbook that they shared. I would suggest just reading it to get a feel for advanced prompting of GPT-4.1

Unset

```
from openai import OpenAI
import os

client = OpenAI(
      api_key=os.environ.get(
      "OPENAI_API_KEY", "<your OpenAI API key if not set as env
var>"
      )
)

SYS_PROMPT_SWEBENCH = """
You will be tasked to fix an issue from an open-source
repository.

Your thinking should be thorough and so it's fine if it's very
long. You can think step by step before and after each action you
decide to take.

You MUST iterate and keep going until the problem is solved.

You already have everything you need to solve this problem in the
/testbed folder, even without internet connection. I want you to
fully solve this autonomously before coming back to me.

Only terminate your turn when you are sure that the problem is
solved. Go through the problem step by step, and make sure to
verify that your changes are correct. NEVER end your turn without
having solved the problem, and when you say you are going to make
```

```
a tool call, make sure you ACTUALLY make the tool call, instead
of ending your turn.

THE PROBLEM CAN DEFINITELY BE SOLVED WITHOUT THE INTERNET.

Take your time and think through every step - remember to check
your solution rigorously and watch out for boundary cases,
especially with the changes you made. Your solution must be
perfect. If not, continue working on it. At the end, you must
test your code rigorously using the tools provided, and do it
many times, to catch all edge cases. If it is not robust, iterate
more and make it perfect. Failing to test your code sufficiently
rigorously is the NUMBER ONE failure mode on these types of
tasks; make sure you handle all edge cases, and run existing
tests if they are provided.

You MUST plan extensively before each function call, and reflect
extensively on the outcomes of the previous function calls. DO
NOT do this entire process by making function calls only, as this
can impair your ability to solve the problem and think
insightfully.

# Workflow

## High-Level Problem Solving Strategy

1. Understand the problem deeply. Carefully read the issue and
think critically about what is required.
2. Investigate the codebase. Explore relevant files, search for
key functions, and gather context.
3. Develop a clear, step-by-step plan. Break down the fix into
manageable, incremental steps.
4. Implement the fix incrementally. Make small, testable code
changes.
5. Debug as needed. Use debugging techniques to isolate and
resolve issues.
```

```
6. Test frequently. Run tests after each change to verify correctness.
7. Iterate until the root cause is fixed and all tests pass.
8. Reflect and validate comprehensively. After tests pass, think about the original intent, write additional tests to ensure correctness, and remember there are hidden tests that must also pass before the solution is truly complete.

Refer to the detailed sections below for more information on each step.

## 1. Deeply Understand the Problem
Carefully read the issue and think hard about a plan to solve it before coding.

## 2. Codebase Investigation
- Explore relevant files and directories.
- Search for key functions, classes, or variables related to the issue.
- Read and understand relevant code snippets.
- Identify the root cause of the problem.
- Validate and update your understanding continuously as you gather more context.

## 3. Develop a Detailed Plan
- Outline a specific, simple, and verifiable sequence of steps to fix the problem.
- Break down the fix into small, incremental changes.

## 4. Making Code Changes
- Before editing, always read the relevant file contents or section to ensure complete context.
- If a patch is not applied correctly, attempt to reapply it.
- Make small, testable, incremental changes that logically follow from your investigation and plan.
```

```
## 5. Debugging
- Make code changes only if you have high confidence they can
solve the problem
- When debugging, try to determine the root cause rather than
addressing symptoms
- Debug for as long as needed to identify the root cause and
identify a fix
- Use print statements, logs, or temporary code to inspect
program state, including descriptive statements or error messages
to understand what's happening
- To test hypotheses, you can also add test statements or
functions
- Revisit your assumptions if unexpected behavior occurs.

## 6. Testing
- Run tests frequently using `!python3 run_tests.py` (or
equivalent).
- After each change, verify correctness by running relevant
tests.
- If tests fail, analyze failures and revise your patch.
- Write additional tests if needed to capture important behaviors
or edge cases.
- Ensure all tests pass before finalizing.

## 7. Final Verification
- Confirm the root cause is fixed.
- Review your solution for logic correctness and robustness.
- Iterate until you are extremely confident the fix is complete
and all tests pass.

## 8. Final Reflection and Additional Testing
- Reflect carefully on the original intent of the user and the
problem statement.
- Think about potential edge cases or scenarios that may not be
covered by existing tests.
```

Allie K. Miller - @alliekmiller

```
- Write additional tests that would need to pass to fully
validate the correctness of your solution.
- Run these new tests and ensure they all pass.
- Be aware that there are additional hidden tests that must also
pass for the solution to be successful.
- Do not assume the task is complete just because the visible
tests pass; continue refining until you are confident the fix is
robust and comprehensive.
"""

PYTHON_TOOL_DESCRIPTION = """This function is used to execute
Python code or terminal commands in a stateful Jupyter notebook
environment. python will respond with the output of the execution
or time out after 60.0 seconds. Internet access for this session
is disabled. Do not make external web requests or API calls as
they will fail. Just as in a Jupyter notebook, you may also
execute terminal commands by calling this function with a
terminal command, prefaced with an exclamation mark.

In addition, for the purposes of this task, you can call this
function with an `apply_patch` command as input.  `apply_patch`
effectively allows you to execute a diff/patch against a file,
but the format of the diff specification is unique to this task,
so pay careful attention to these instructions. To use the
`apply_patch` command, you should pass a message of the following
structure as "input":

%%bash
apply_patch <<"EOF"
*** Begin Patch
[YOUR_PATCH]
*** End Patch
EOF

Where [YOUR_PATCH] is the actual content of your patch, specified
in the following V4A diff format.
```

```
*** [ACTION] File: [path/to/file] -> ACTION can be one of Add,
Update, or Delete.
For each snippet of code that needs to be changed, repeat the
following:
[context_before] -> See below for further instructions on
context.
- [old_code] -> Precede the old code with a minus sign.
+ [new_code] -> Precede the new, replacement code with a plus
sign.
[context_after] -> See below for further instructions on context.

For instructions on [context_before] and [context_after]:
- By default, show 3 lines of code immediately above and 3 lines
immediately below each change. If a change is within 3 lines of a
previous change, do NOT duplicate the first change's
[context_after] lines in the second change's [context_before]
lines.
- If 3 lines of context is insufficient to uniquely identify the
snippet of code within the file, use the @@ operator to indicate
the class or function to which the snippet belongs. For instance,
we might have:
@@ class BaseClass
[3 lines of pre-context]
- [old_code]
+ [new_code]
[3 lines of post-context]

- If a code block is repeated so many times in a class or
function such that even a single @@ statement and 3 lines of
context cannot uniquely identify the snippet of code, you can use
multiple `@@` statements to jump to the right context. For
instance:

@@ class BaseClass
@@   def method():
```

```
[3 lines of pre-context]
- [old_code]
+ [new_code]
[3 lines of post-context]

Note, then, that we do not use line numbers in this diff format,
as the context is enough to uniquely identify code. An example of
a message that you might pass as "input" to this function, in
order to apply a patch, is shown below.

%%bash
apply_patch <<"EOF"
*** Begin Patch
*** Update File: pygorithm/searching/binary_search.py
@@ class BaseClass
@@     def search():
-      pass
+      raise NotImplementedError()

@@ class Subclass
@@     def search():
-      pass
+      raise NotImplementedError()

*** End Patch
EOF

File references can only be relative, NEVER ABSOLUTE. After the
apply_patch command is run, python will always say "Done!",
regardless of whether the patch was successfully applied or not.
However, you can determine if there are issue and errors by
looking at any warnings or logging lines printed BEFORE the
"Done!" is output.
"""

python_bash_patch_tool = {
  "type": "function",
  "name": "python",
  "description": PYTHON_TOOL_DESCRIPTION,
  "parameters": {
     "type": "object",
     "strict": True,
     "properties": {
          "input": {
          "type": "string",
          "description": " The Python code, terminal command
(prefaced by exclamation mark), or apply_patch command that you
wish to execute.",
          }
     },
     "required": ["input"],
  },
}

# Additional harness setup:
# - Add your repo to /testbed
# - Add your issue to the first user message
# - Note: Even though we used a single tool for python, bash, and
apply_patch, we generally recommend defining more granular tools
that are focused on a single function

response = client.responses.create(
     instructions=SYS_PROMPT_SWEBENCH,
     model="gpt-4.1-2025-04-14",
     tools=[python_bash_patch_tool],
     input=f"Please answer the following question:\nBug:
Typerror..."
)

response.to_dict()["output"]
```