Apr 14, 2025

GPT-4.1 Prompting Guide

Noah MacCallum (OpenAI),
Julian Lee (OpenAI)

Open in Github

View as Markdown

The GPT-4.1 family of models represents a significant step forward from GPT-4o in capabilities across coding, instruction following, and long context. In this prompting guide, we collate a series of important prompting tips derived from extensive internal testing to help developers fully leverage the improved abilities of this new model family.

Many typical best practices still apply to GPT-4.1, such as providing context examples, making instructions as specific and clear as possible, and inducing planning via prompting to maximize model intelligence. However, we expect that getting the most out of this model will require some prompt migration. GPT-4.1 is trained to follow instructions more closely and more literally than its predecessors, which tended to more liberally infer intent from user and system prompts. This also means, however, that GPT-4.1 is highly steerable and responsive to well-specified prompts - if model behavior is different from what you expect, a single sentence firmly and unequivocally clarifying your desired behavior is almost always sufficient to steer the model on course.

Please read on for prompt examples you can use as a reference, and remember that while this guidance is widely applicable, no advice is one-size-fits-all. AI engineering is inherently an empirical discipline, and large language models are inherently nondeterministic; in addition to following this guide, we advise building informative evals and iterating often to ensure your prompt engineering changes are yielding benefits for your use case.

1. Agentic Workflows

GPT-4.1 is a great place to build agentic workflows. In model training we emphasized providing a diverse range of agentic problem-solving trajectories, and our agentic harness for the model achieves state-of-the-art performance for non-reasoning models on SWE-bench Verified, solving 55% of problems.

System Prompt Reminders

In order to fully utilize the agentic capabilities of GPT-4.1, we recommend including three key types of reminders in all agent prompts. The following prompts are optimized specifically for the agentic coding workflow, but can be easily modified for general agentic use cases.

1. Persistence: this ensures the model understands it is entering a multi-message turn, and prevents it from prematurely yielding control back to the user. Our example is the following:

```
You are an agent - please keep going until the user’s query is completely resolved, bef
```

2. Tool-calling: this encourages the model to make full use of its tools, and reduces its likelihood of hallucinating or guessing an answer. Our example is the following:

```
If you are not sure about file content or codebase structure pertaining to the user’s r
```

3. Planning [optional]: if desired, this ensures the model explicitly plans and reflects upon each tool call in text, instead of completing the task by chaining together a series of only tool calls. Our example is the following:

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

GPT-4.1 is trained to respond very closely to both user instructions and system prompts in the agentic setting. The model adhered closely to these three simple instructions and increased our internal SWE-bench Verified score by close to 20% - so we highly encourage starting any agent prompt with clear reminders covering the three categories listed above. As a whole, we find that these three instructions transform the model from a chatbot-like state into a much more “eager” agent, driving the interaction forward autonomously and independently.

Tool Calls

Compared to previous models, GPT-4.1 has undergone more training on effectively utilizing tools passed as arguments in an OpenAI API request. We encourage developers to exclusively use the tools field to pass tools, rather than manually injecting tool descriptions into your prompt and writing a separate parser for tool calls, as some have reported doing in the past. This is the best way to minimize errors and ensure the model remains in distribution during tool-calling trajectories - in our own experiments, we observed a 2% increase in SWE-bench Verified pass rate when using API-parsed tool descriptions versus manually injecting the schemas into the system prompt.

Developers should name tools clearly to indicate their purpose and add a clear, detailed description in the "description" field of the tool. Similarly, for each tool param, lean on good naming and descriptions to ensure appropriate usage. If your tool is particularly complicated and you'd like to provide examples of tool usage, we recommend that you create an `# Examples` section in your system prompt and place the examples there, rather than adding them into the "description' field, which should remain thorough but relatively concise. Providing examples can be helpful to indicate when to use tools, whether to include user text alongside tool calls, and what parameters are appropriate for different inputs. Remember that you can use “Generate Anything” in the Prompt Playground to get a good starting point for your new tool definitions.

Prompting-Induced Planning & Chain-of-Thought

As mentioned already, developers can optionally prompt agents built with GPT-4.1 to plan and reflect between tool calls, instead of silently calling tools in an unbroken sequence. GPT-4.1 is not a reasoning model - meaning that it does not produce an internal chain of thought before answering - but in the prompt, a developer can induce the model to produce an explicit, step-by-step plan by using any variant of the Planning prompt component shown above. This can be thought of as the model “thinking out loud.” In our experimentation with the SWE-bench Verified agentic task, inducing explicit planning increased the pass rate by 4%.

Sample Prompt: SWE-bench Verified

Below, we share the agentic prompt that we used to achieve our highest score on SWE-bench Verified, which features detailed instructions about workflow and problem-solving strategy. This general pattern can be used for any agentic task.

```
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

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,

Only terminate your turn when you are sure that the problem is solved. Go through t

THE PROBLEM CAN DEFINITELY BE SOLVED WITHOUT THE INTERNET.

Take your time and think through every step - remember to check your solution rigor

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

# Workflow

## High-Level Problem Solving Strategy

1. Understand the problem deeply. Carefully read the issue and think critically abo
2. Investigate the codebase. Explore relevant files, search for key functions, and
3. Develop a clear, step-by-step plan. Break down the fix into manageable, incremen
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

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 compl
- If a patch is not applied correctly, attempt to reapply it.
- Make small, testable, incremental changes that logically follow from your investi

## 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
- 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
- Write additional tests that would need to pass to fully validate the correctness
- Run these new tests and ensure they all pass.
- Be aware that there are additional hidden tests that must also pass for the solut
- Do not assume the task is complete just because the visible tests pass; continue
"""

PYTHON_TOOL_DESCRIPTION = """This function is used to execute Python code or termin

In addition, for the purposes of this task, you can call this function with an `app

%%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

*** [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
- If 3 lines of context is insufficient to uniquely identify the snippet of code wi
@@ 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

@@ 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

%%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
"""

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 excla
          }
      },
      "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 g

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"]
```

```
[{'id': 'msg_67fe92df26ac819182ffafce9ff4e4fc07c7e06242e51f8b',
  'content': [{'annotations': [],
    'text': "Thank you for the report, but “Typerror” is too vague for me to start
    'type': 'output_text'}],
  'role': 'assistant',
  'status': 'completed',
  'type': 'message'},
 {'arguments': '{"input":"!ls -l /testbed"}',
  'call_id': 'call_frnxyJgKi5TsBem0nR9Zuzdw',
  'name': 'python',
  'type': 'function_call',
  'id': 'fc_67fe92e3da7081918fc18d5c96dddc1c07c7e06242e51f8b',
  'status': 'completed'}]
```

2. Long context

GPT-4.1 has a performant 1M token input context window, and is useful for a variety of long context tasks, including structured document parsing, re-ranking, selecting relevant information while ignoring irrelevant context, and performing multi-hop reasoning using context.

Optimal Context Size

We observe very good performance on needle-in-a-haystack evaluations up to our full 1M token context, and we've observed very strong performance at complex tasks with a mix of both relevant and irrelevant code and other documents. However, long context performance can degrade as more items are required to be retrieved, or perform complex reasoning that requires knowledge of the state of the entire context (like performing a graph search, for example).

Tuning Context Reliance

Consider the mix of external vs. internal world knowledge that might be required to answer your question. Sometimes it's important for the model to use some of its own knowledge to connect concepts or make logical jumps, while in others it's desirable to only use provided context

```
# Instructions
// for internal knowledge
- Only use the documents in the provided External Context to answer the User Query. If
// For internal and external knowledge
- By default, use the provided external context to answer the User Query, but if other
```

Prompt Organization

Especially in long context usage, placement of instructions and context can impact performance. If you have long context in your prompt, ideally place your instructions at both the beginning and end of the provided context, as we found this to perform better than only above or below. If you'd prefer to only have your instructions once, then above the provided context works better than below.

3. Chain of Thought

As mentioned above, GPT-4.1 is not a reasoning model, but prompting the model to think step by step (called "chain of thought") can be an effective way for a model to break down problems into more manageable pieces, solve them, and improve overall output quality, with the tradeoff of higher cost and latency associated with using more output tokens. The model has been trained to perform well at agentic reasoning about and real-world problem solving, so it shouldn't require much prompting to perform well.

We recommend starting with this basic chain-of-thought instruction at the end of your prompt:

```
...
First, think carefully step by step about what documents are needed to answer the query
```

From there, you should improve your chain-of-thought (CoT) prompt by auditing failures in your particular examples and evals, and addressing systematic planning and reasoning errors with more explicit instructions. In the unconstrained CoT prompt, there may be variance in the strategies it tries, and if you observe an approach that works well, you can codify that strategy in your prompt. Generally speaking, errors tend to occur from misunderstanding user intent, insufficient context gathering or analysis, or insufficient or incorrect step by step thinking, so watch out for these and try to address them with more opinionated instructions.

Here is an example prompt instructing the model to focus more methodically on analyzing user intent and considering relevant context before proceeding to answer.

```
# Reasoning Strategy
1. Query Analysis: Break down and analyze the query until you're confident about what i
2. Context Analysis: Carefully select and analyze a large set of potentially relevant c
    a. Analysis: An analysis of how it may or may not be relevant to answering the quer
    b. Relevance rating: [high, medium, low, none]
3. Synthesis: summarize which documents are most relevant and why, including all docume
# User Question
{user_question}
# External Context
{external_context}
First, think carefully step by step about what documents are needed to answer the query
```

4. Instruction Following

GPT-4.1 exhibits outstanding instruction-following performance, which developers can leverage to precisely shape and control the outputs for their particular use cases. Developers often extensively prompt for agentic reasoning steps, response tone and voice, tool calling information, output formatting, topics to avoid, and more. However, since the model follows instructions more literally, developers may need to include explicit specification around what to do or not to do. Furthermore, existing prompts optimized for other models may not immediately work with this model, because existing instructions are followed more closely and implicit rules are no longer being as strongly inferred.

Recommended Workflow

Here is our recommended workflow for developing and debugging instructions in prompts:

1. Start with an overall “Response Rules” or “Instructions” section with high-level guidance and bullet points.
2. If you’d like to change a more specific behavior, add a section to specify more details for that category, like `# Sample Phrases`.
3. If there are specific steps you’d like the model to follow in its workflow, add an ordered list and instruct the model to follow these steps.
4. If behavior still isn’t working as expected:
    1. Check for conflicting, underspecified, or wrong instructions and examples. If there are conflicting instructions, GPT-4.1 tends to follow the one closer to the end of the prompt.
    2. Add examples that demonstrate desired behavior; ensure that any important behavior demonstrated in your examples are also cited in your rules.
    3. It’s generally not necessary to use all-caps or other incentives like bribes or tips. We recommend starting without these, and only reaching for these if necessary for your particular prompt. Note that if your existing prompts include these techniques, it could cause GPT-4.1 to pay attention to it too strictly.

Note that using your preferred AI-powered IDE can be very helpful for iterating on prompts, including checking for consistency or conflicts, adding examples, or making cohesive updates like adding an instruction and updating instructions to demonstrate that instruction.

Common Failure Modes

These failure modes are not unique to GPT-4.1, but we share them here for general awareness and ease of debugging.

- Instructing a model to always follow a specific behavior can occasionally induce adverse effects. For instance, if told “you must call a tool before responding to the user,” models may hallucinate tool inputs or call the tool with null values if they do not have enough information. Adding “if you don’t have enough information to call the tool, ask the user for the information you need” should mitigate this.

- When provided sample phrases, models can use those quotes verbatim and start to sound repetitive to users. Ensure you instruct the model to vary them as necessary.
- Without specific instructions, some models can be eager to provide additional prose to explain their decisions, or output more formatting in responses than may be desired. Provide instructions and potentially examples to help mitigate.

Example Prompt: Customer Service

This demonstrates best practices for a fictional customer service agent. Observe the diversity of rules, the specificity, the use of additional sections for greater detail, and an example to demonstrate precise behavior that incorporates all prior rules.

Try running the following notebook cell - you should see both a user message and tool call, and the user message should start with a greeting, then echo back their answer, then mention they're about to call a tool. Try changing the instructions to shape the model behavior, or trying other user messages, to test instruction following performance.

```
SYS_PROMPT_CUSTOMER_SERVICE = """You are a helpful customer service agent working f

# Instructions
- Always greet the user with "Hi, you've reached NewTelco, how can I help you?"
- Always call a tool before answering factual questions about the company, its offe
    - However, if you don't have enough information to properly call the tool, ask
- Escalate to a human if the user requests.
- Do not discuss prohibited topics (politics, religion, controversial current event
- Rely on sample phrases whenever appropriate, but never repeat a sample phrase in
- Always follow the provided output format for new messages, including citations fo
- If you're going to call a tool, always message the user with an appropriate messa
- Maintain a professional and concise tone in all responses, and use emojis between
- If you've resolved the user's request, ask if there's anything else you can help

# Precise Response Steps (for each response)
1. If necessary, call tools to fulfill the user's desired action. Always message th
2. In your response to the user
    a. Use active listening and echo back what you heard the user ask for.
    b. Respond appropriately given the above guidelines.

# Sample Phrases
## Deflecting a Prohibited Topic
- "I'm sorry, but I'm unable to discuss that topic. Is there something else I can h
- "That's not something I'm able to provide information on, but I'm happy to help w
```

```
## Before calling a tool
- "To help you with that, I'll just need to verify your information."
- "Let me check that for you—one moment, please."
- "I'll retrieve the latest details for you now."

## After calling a tool
- "Okay, here's what I found: [response]"
- "So here's what I found: [response]"

# Output Format
- Always include your final response to the user.
- When providing factual information from retrieved context, always include citatio
    - For a single source: [NAME](ID)
    - For multiple sources: [NAME](ID), [NAME](ID)
- Only provide information about this company, its policies, its products, or the c

# Example
## User
Can you tell me about your family plan options?

## Assistant Response 1
### Message
"Hi, you've reached NewTelco, how can I help you? 😊🎉\n\nYou'd like to know about

### Tool Calls
lookup_policy_document(topic="family plan options")

// After tool call, the assistant would follow up with:

## Assistant Response 2 (after tool call)
### Message
"Okay, here's what I found: 🎉 Our family plan allows up to 5 lines with shared dat
"""

get_policy_doc = {
    "type": "function",
    "name": "lookup_policy_document",
    "description": "Tool to look up internal documents and policies by topic or key
    "parameters": {
        "strict": True,
        "type": "object",
        "properties": {
            "topic": {
                "type": "string",
                "description": "The topic or keyword to search for in company polic
            },
        },
        "required": ["topic"],
        "additionalProperties": False,
    },
}
```

```
get_user_acct = {
    "type": "function",
    "name": "get_user_account_info",
    "description": "Tool to get user account information",
    "parameters": {
        "strict": True,
        "type": "object",
        "properties": {
            "phone_number": {
                "type": "string",
                "description": "Formatted as '(xxx) xxx-xxxx'",
            },
        },
        "required": ["phone_number"],
        "additionalProperties": False,
    },
}

response = client.responses.create(
    instructions=SYS_PROMPT_CUSTOMER_SERVICE,
    model="gpt-4.1-2025-04-14",
    tools=[get_policy_doc, get_user_acct],
    input="How much will it cost for international service? I'm traveling to France
    # input="Why was my last bill so high?"
)

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

```
[{'id': 'msg_67fe92d431548191b7ca6cd604b4784b06efc5beb16b3c5e',
  'content': [{'annotations': [],
    'text': "Hi, you've reached NewTelco, how can I help you? 🌍✈️\n\nYou'd like to
    'type': 'output_text'}],
  'role': 'assistant',
  'status': 'completed',
  'type': 'message'},
 {'arguments': '{"topic":"international service cost France"}',
  'call_id': 'call_cF63DLeyhNhwfdyME3ZHd0yo',
  'name': 'lookup_policy_document',
  'type': 'function_call',
  'id': 'fc_67fe92d5d6888191b6cd7cf57f707e4606efc5beb16b3c5e',
  'status': 'completed'}]
```

5. General Advice

Prompt Structure

For reference, here is a good starting point for structuring your prompts.

```
# 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
```

Add or remove sections to suit your needs, and experiment to determine what's optimal for your usage.

Delimiters

Here are some general guidelines for selecting the best delimiters for your prompt. Please refer to the Long Context section for special considerations for that context type.

1. Markdown: We recommend starting here, and using markdown titles for major sections and subsections (including deeper hierarchy, to H4+). Use inline backticks or backtick blocks to precisely wrap code, and standard numbered or bulleted lists as needed.
2. XML: These also perform well, and we have improved adherence to information in XML with this model. XML is convenient to precisely wrap a section including start and end, add metadata to the tags for additional context, and enable nesting. Here is an example of using XML tags to nest examples in an example section, with inputs and outputs for each:

```
<examples>
<example1 type="Abbreviate">
<input>San Francisco</input>
<output>- SF</output>
</example1>
</examples>
```

3. JSON is highly structured and well understood by the model particularly in coding contexts. However it can be more verbose, and require character escaping that can add overhead.

Guidance specifically for adding a large number of documents or files to input context:

- XML performed well in our long context testing.
  - Example: `<doc id='1' title='The Fox'>The quick brown fox jumps over the lazy dog</doc>`
- This format, proposed by Lee et al. (ref), also performed well in our long context testing.
  - Example: `ID: 1 | TITLE: The Fox | CONTENT: The quick brown fox jumps over the lazy dog`
- JSON performed particularly poorly.
  - Example: `[{'id': 1, 'title': 'The Fox', 'content': 'The quick brown fox jumped over the lazy dog'}]`

The model is trained to robustly understand structure in a variety of formats. Generally, use your judgement and think about what will provide clear information and "stand out" to the model. For example, if you're retrieving documents that contain lots of XML, an XML-based delimiter will likely be less effective.

Caveats

- In some isolated cases we have observed the model being resistant to producing very long, repetitive outputs, for example, analyzing hundreds of items one by one. If this is necessary for your use case, instruct the model strongly to output this information in full, and consider breaking down the problem or using a more concise approach.
- We have seen some rare instances of parallel tool calls being incorrect. We advise testing this, and considering setting the parallel_tool_calls param to false if you're seeing issues.

Appendix: Generating and Applying File Diffs

Developers have provided us feedback that accurate and well-formed diff generation is a critical capability to power coding-related tasks. To this end, the GPT-4.1 family features substantially improved diff capabilities relative to previous GPT models. Moreover, while GPT-4.1 has strong performance generating diffs of any format given clear instructions and examples, we open-source here one recommended diff format, on which the model has been extensively trained. We hope that in particular for developers just starting out, that this will take much of the guesswork out of creating diffs yourself.

Apply Patch

See the example below for a prompt that applies our recommended tool call correctly.

```
APPLY_PATCH_TOOL_DESC = """This is a custom utility that makes it more convenient t

%%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

*** [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
- If 3 lines of context is insufficient to uniquely identify the snippet of code wi
@@ 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

@@ 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

%%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
"""

APPLY_PATCH_TOOL = {
    "name": "apply_patch",
    "description": APPLY_PATCH_TOOL_DESC,
    "parameters": {
        "type": "object",
        "properties": {
            "input": {
                "type": "string",
                "description": " The apply_patch command that you wish to execute."
            }
        },
        "required": ["input"],
    },
}
```

Reference Implementation: apply_patch.py

Here’s a reference implementation of the apply_patch tool that we used as part of model training. You’ll need to make this an executable and available as `apply_patch` from the shell where the model will execute commands:

```
#!/usr/bin/env python3

"""
A self-contained **pure-Python 3.9+** utility for applying human-readable
“pseudo-diff” patch files to a collection of text files.
"""

from __future__ import annotations

import pathlib
from dataclasses import dataclass, field
from enum import Enum
from typing import (
    Callable,
    Dict,
    List,
    Optional,
    Tuple,
    Union,
)


# --------------------------------------------------------------------------- #
#  Domain objects
# --------------------------------------------------------------------------- #
class ActionType(str, Enum):
    ADD = "add"
    DELETE = "delete"
    UPDATE = "update"


@dataclass
class FileChange:
    type: ActionType
    old_content: Optional[str] = None
    new_content: Optional[str] = None
    move_path: Optional[str] = None


@dataclass
class Commit:
    changes: Dict[str, FileChange] = field(default_factory=dict)


# --------------------------------------------------------------------------- #
#  Exceptions
# --------------------------------------------------------------------------- #
class DiffError(ValueError):
    """Any problem detected while parsing or applying a patch."""


# --------------------------------------------------------------------------- #
#  Helper dataclasses used while parsing patches
# --------------------------------------------------------------------------- #
@dataclass
class Chunk:
    orig_index: int = -1
    del_lines: List[str] = field(default_factory=list)
    ins_lines: List[str] = field(default_factory=list)


@dataclass
class PatchAction:
    type: ActionType
    new_file: Optional[str] = None
    chunks: List[Chunk] = field(default_factory=list)
    move_path: Optional[str] = None


@dataclass
class Patch:
    actions: Dict[str, PatchAction] = field(default_factory=dict)


# --------------------------------------------------------------------------- #
#  Patch text parser
# --------------------------------------------------------------------------- #
@dataclass
class Parser:
    current_files: Dict[str, str]
    lines: List[str]
    index: int = 0
    patch: Patch = field(default_factory=Patch)
    fuzz: int = 0

    # ------------- low-level helpers --------------------------------------- #
    def _cur_line(self) -> str:
        if self.index >= len(self.lines):
            raise DiffError("Unexpected end of input while parsing patch")
        return self.lines[self.index]

    @staticmethod
    def _norm(line: str) -> str:
        """Strip CR so comparisons work for both LF and CRLF input."""
        return line.rstrip("\r")

    # ------------- scanning convenience ------------------------------------ #
    def is_done(self, prefixes: Optional[Tuple[str, ...]] = None) -> bool:
        if self.index >= len(self.lines):
            return True
        if (
            prefixes
            and len(prefixes) > 0
            and self._norm(self._cur_line()).startswith(prefixes)
        ):
            return True
        return False

    def startswith(self, prefix: Union[str, Tuple[str, ...]]) -> bool:
        return self._norm(self._cur_line()).startswith(prefix)

    def read_str(self, prefix: str) -> str:
        """
        Consume the current line if it starts with *prefix* and return the text
        **after** the prefix.  Raises if prefix is empty.
        """
        if prefix == "":
            raise ValueError("read_str() requires a non-empty prefix")
        if self._norm(self._cur_line()).startswith(prefix):
            text = self._cur_line()[len(prefix) :]
            self.index += 1
            return text
        return ""

    def read_line(self) -> str:
        """Return the current raw line and advance."""
        line = self._cur_line()
        self.index += 1
        return line

    # ------------- public entry point -------------------------------------- #
    def parse(self) -> None:
        while not self.is_done(("*** End Patch",)):
            # ---------- UPDATE ---------- #
            path = self.read_str("*** Update File: ")
            if path:
                if path in self.patch.actions:
                    raise DiffError(f"Duplicate update for file: {path}")
                move_to = self.read_str("*** Move to: ")
                if path not in self.current_files:
                    raise DiffError(f"Update File Error - missing file: {path}")
                text = self.current_files[path]
                action = self._parse_update_file(text)
                action.move_path = move_to or None
                self.patch.actions[path] = action
                continue

            # ---------- DELETE ---------- #
            path = self.read_str("*** Delete File: ")
            if path:
                if path in self.patch.actions:
                    raise DiffError(f"Duplicate delete for file: {path}")
                if path not in self.current_files:
                    raise DiffError(f"Delete File Error - missing file: {path}")
                self.patch.actions[path] = PatchAction(type=ActionType.DELETE)
                continue

            # ---------- ADD ---------- #
            path = self.read_str("*** Add File: ")
            if path:
                if path in self.patch.actions:
                    raise DiffError(f"Duplicate add for file: {path}")
                if path in self.current_files:
                    raise DiffError(f"Add File Error - file already exists: {path}"
                self.patch.actions[path] = self._parse_add_file()
                continue
            raise DiffError(f"Unknown line while parsing: {self._cur_line()}")

        if not self.startswith("*** End Patch"):
            raise DiffError("Missing *** End Patch sentinel")
        self.index += 1  # consume sentinel

    # ------------- section parsers ---------------------------------------- #
    def _parse_update_file(self, text: str) -> PatchAction:
        action = PatchAction(type=ActionType.UPDATE)
        lines = text.split("\n")
        index = 0
        while not self.is_done(
            (
                "*** End Patch",
                "*** Update File:",
                "*** Delete File:",
                "*** Add File:",
                "*** End of File",
            )
        ):
            def_str = self.read_str("@@ ")
            section_str = ""
            if not def_str and self._norm(self._cur_line()) == "@@":
                section_str = self.read_line()

            if not (def_str or section_str or index == 0):
                raise DiffError(f"Invalid line in update section:\n{self._cur_line(

            if def_str.strip():
                found = False
                if def_str not in lines[:index]:
                    for i, s in enumerate(lines[index:], index):
                        if s == def_str:
                            index = i + 1
                            found = True
                            break
                if not found and def_str.strip() not in [
                    s.strip() for s in lines[:index]
                ]:
                    for i, s in enumerate(lines[index:], index):
                        if s.strip() == def_str.strip():
                            index = i + 1
                            self.fuzz += 1
                            found = True
                            break

            next_ctx, chunks, end_idx, eof = peek_next_section(self.lines, self.ind
            new_index, fuzz = find_context(lines, next_ctx, index, eof)
            if new_index == -1:
                ctx_txt = "\n".join(next_ctx)
                raise DiffError(
                    f"Invalid {'EOF ' if eof else ''}context at {index}:\n{ctx_txt}
                )
            self.fuzz += fuzz
            for ch in chunks:
                ch.orig_index += new_index
                action.chunks.append(ch)
            index = new_index + len(next_ctx)
            self.index = end_idx
        return action

    def _parse_add_file(self) -> PatchAction:
        lines: List[str] = []
        while not self.is_done(
            ("*** End Patch", "*** Update File:", "*** Delete File:", "*** Add File
        ):
            s = self.read_line()
            if not s.startswith("+"):
                raise DiffError(f"Invalid Add File line (missing '+'): {s}")
            lines.append(s[1:])  # strip leading '+'
        return PatchAction(type=ActionType.ADD, new_file="\n".join(lines))


# --------------------------------------------------------------------------- #
#  Helper functions
# --------------------------------------------------------------------------- #
def find_context_core(
    lines: List[str], context: List[str], start: int
) -> Tuple[int, int]:
    if not context:
        return start, 0

    for i in range(start, len(lines)):
        if lines[i : i + len(context)] == context:
            return i, 0
    for i in range(start, len(lines)):
        if [s.rstrip() for s in lines[i : i + len(context)]] == [
            s.rstrip() for s in context
        ]:
            return i, 1
    for i in range(start, len(lines)):
        if [s.strip() for s in lines[i : i + len(context)]] == [
            s.strip() for s in context
        ]:
            return i, 100
    return -1, 0


def find_context(
    lines: List[str], context: List[str], start: int, eof: bool
) -> Tuple[int, int]:
    if eof:
        new_index, fuzz = find_context_core(lines, context, len(lines) - len(contex
        if new_index != -1:
            return new_index, fuzz
        new_index, fuzz = find_context_core(lines, context, start)
        return new_index, fuzz + 10_000
    return find_context_core(lines, context, start)


def peek_next_section(
    lines: List[str], index: int
) -> Tuple[List[str], List[Chunk], int, bool]:
    old: List[str] = []
    del_lines: List[str] = []
    ins_lines: List[str] = []
    chunks: List[Chunk] = []
    mode = "keep"
    orig_index = index

    while index < len(lines):
        s = lines[index]
        if s.startswith(
            (
                "@@",
                "*** End Patch",
                "*** Update File:",
                "*** Delete File:",
                "*** Add File:",
                "*** End of File",
            )
        ):
            break
        if s == "***":
            break
        if s.startswith("***"):
            raise DiffError(f"Invalid Line: {s}")
        index += 1

        last_mode = mode
        if s == "":
            s = " "
        if s[0] == "+":
            mode = "add"
        elif s[0] == "-":
            mode = "delete"
        elif s[0] == " ":
            mode = "keep"
        else:
            raise DiffError(f"Invalid Line: {s}")
        s = s[1:]

        if mode == "keep" and last_mode != mode:
            if ins_lines or del_lines:
                chunks.append(
                    Chunk(
                        orig_index=len(old) - len(del_lines),
                        del_lines=del_lines,
                        ins_lines=ins_lines,
                    )
                )
            del_lines, ins_lines = [], []

        if mode == "delete":
            del_lines.append(s)
            old.append(s)
        elif mode == "add":
            ins_lines.append(s)
        elif mode == "keep":
            old.append(s)

    if ins_lines or del_lines:
        chunks.append(
            Chunk(
                orig_index=len(old) - len(del_lines),
                del_lines=del_lines,
                ins_lines=ins_lines,
            )
        )

    if index < len(lines) and lines[index] == "*** End of File":
        index += 1
        return old, chunks, index, True

    if index == orig_index:
        raise DiffError("Nothing in this section")
    return old, chunks, index, False


# --------------------------------------------------------------------------- #
#  Patch → Commit and Commit application
# --------------------------------------------------------------------------- #
def _get_updated_file(text: str, action: PatchAction, path: str) -> str:
    if action.type is not ActionType.UPDATE:
        raise DiffError("_get_updated_file called with non-update action")
    orig_lines = text.split("\n")
    dest_lines: List[str] = []
    orig_index = 0

    for chunk in action.chunks:
        if chunk.orig_index > len(orig_lines):
            raise DiffError(
                f"{path}: chunk.orig_index {chunk.orig_index} exceeds file length"
            )
        if orig_index > chunk.orig_index:
            raise DiffError(
                f"{path}: overlapping chunks at {orig_index} > {chunk.orig_index}"
            )
        dest_lines.extend(orig_lines[orig_index : chunk.orig_index])
        orig_index = chunk.orig_index

        dest_lines.extend(chunk.ins_lines)
        orig_index += len(chunk.del_lines)

    dest_lines.extend(orig_lines[orig_index:])
    return "\n".join(dest_lines)


def patch_to_commit(patch: Patch, orig: Dict[str, str]) -> Commit:
    commit = Commit()
    for path, action in patch.actions.items():
        if action.type is ActionType.DELETE:
            commit.changes[path] = FileChange(
                type=ActionType.DELETE, old_content=orig[path]
            )
        elif action.type is ActionType.ADD:
            if action.new_file is None:
                raise DiffError("ADD action without file content")
            commit.changes[path] = FileChange(
                type=ActionType.ADD, new_content=action.new_file
            )
        elif action.type is ActionType.UPDATE:
            new_content = _get_updated_file(orig[path], action, path)
            commit.changes[path] = FileChange(
                type=ActionType.UPDATE,
                old_content=orig[path],
                new_content=new_content,
                move_path=action.move_path,
            )
    return commit


# --------------------------------------------------------------------------- #
#  User-facing helpers
# --------------------------------------------------------------------------- #
def text_to_patch(text: str, orig: Dict[str, str]) -> Tuple[Patch, int]:
    lines = text.splitlines()  # preserves blank lines, no strip()
    if (
        len(lines) < 2
        or not Parser._norm(lines[0]).startswith("*** Begin Patch")
        or Parser._norm(lines[-1]) != "*** End Patch"
    ):
        raise DiffError("Invalid patch text - missing sentinels")

    parser = Parser(current_files=orig, lines=lines, index=1)
    parser.parse()
    return parser.patch, parser.fuzz
```

```
def identify_files_needed(text: str) -> List[str]:
    lines = text.splitlines()
    return [
        line[len("*** Update File: ") :]
        for line in lines
        if line.startswith("*** Update File: ")
    ] + [
        line[len("*** Delete File: ") :]
        for line in lines
        if line.startswith("*** Delete File: ")
    ]


def identify_files_added(text: str) -> List[str]:
    lines = text.splitlines()
    return [
        line[len("*** Add File: ") :]
        for line in lines
        if line.startswith("*** Add File: ")
    ]


# --------------------------------------------------------------------------- #
#  File-system helpers
# --------------------------------------------------------------------------- #
def load_files(paths: List[str], open_fn: Callable[[str], str]) -> Dict[str, str]:
    return {path: open_fn(path) for path in paths}


def apply_commit(
    commit: Commit,
    write_fn: Callable[[str, str], None],
    remove_fn: Callable[[str], None],
) -> None:
    for path, change in commit.changes.items():
        if change.type is ActionType.DELETE:
            remove_fn(path)
        elif change.type is ActionType.ADD:
            if change.new_content is None:
                raise DiffError(f"ADD change for {path} has no content")
            write_fn(path, change.new_content)
        elif change.type is ActionType.UPDATE:
            if change.new_content is None:
                raise DiffError(f"UPDATE change for {path} has no new content")
            target = change.move_path or path
            write_fn(target, change.new_content)
            if change.move_path:
                remove_fn(path)


def process_patch(
    text: str,
    open_fn: Callable[[str], str],
    write_fn: Callable[[str, str], None],
    remove_fn: Callable[[str], None],
) -> str:
    if not text.startswith("*** Begin Patch"):
        raise DiffError("Patch text must start with *** Begin Patch")
    paths = identify_files_needed(text)
    orig = load_files(paths, open_fn)
    patch, _fuzz = text_to_patch(text, orig)
    commit = patch_to_commit(patch, orig)
    apply_commit(commit, write_fn, remove_fn)
    return "Done!"


# --------------------------------------------------------------------------- #
#  Default FS helpers
# --------------------------------------------------------------------------- #
def open_file(path: str) -> str:
    with open(path, "rt", encoding="utf-8") as fh:
        return fh.read()


def write_file(path: str, content: str) -> None:
    target = pathlib.Path(path)
    target.parent.mkdir(parents=True, exist_ok=True)
    with target.open("wt", encoding="utf-8") as fh:
        fh.write(content)


def remove_file(path: str) -> None:
    pathlib.Path(path).unlink(missing_ok=True)


# --------------------------------------------------------------------------- #
#  CLI entry-point
# --------------------------------------------------------------------------- #
def main() -> None:
    import sys

    patch_text = sys.stdin.read()
    if not patch_text:
        print("Please pass patch text through stdin", file=sys.stderr)
        return
    try:
        result = process_patch(patch_text, open_file, write_file, remove_file)
    except DiffError as exc:
        print(exc, file=sys.stderr)
        return
    print(result)


if __name__ == "__main__":
    main()
```

Other Effective Diff Formats

If you want to try using a different diff format, we found in testing that the SEARCH/REPLACE diff format used in Aider's polyglot benchmark, as well as a pseudo-XML format with no internal escaping, both had high success rates.

These diff formats share two key aspects: (1) they do not use line numbers, and (2) they provide both the exact code to be replaced, and the exact code with which to replace it, with clear delimiters between the two.

~~~
SEARCH_REPLACE_DIFF_EXAMPLE = """
path/to/file.py
```
>>>>>>> SEARCH
def search():
    pass
=======
def search():
   raise NotImplementedError()
<<<<<<< REPLACE
"""

PSEUDO_XML_DIFF_EXAMPLE = """
<edit>
<file>
path/to/file.py
</file>
<old_code>
def search():
    pass
</old_code>
<new_code>
def search():
   raise NotImplementedError()
</new_code>
</edit>
"""
~~~