GPT-5 prompting guide

GPT-5, our newest flagship model, represents a substantial leap forward in agentic task performance, coding, raw intelligence, and steerability.

While we trust it will perform excellently "out of the box" across a wide range of domains, in this guide we'll cover prompting tips to maximize the quality of model outputs, derived from our experience training and applying the model to real-world tasks. We discuss concepts like improving agentic task performance, ensuring instruction adherence, making use of newly API features, and optimizing coding for frontend and software engineering tasks - with key insights into AI code editor Cursor's prompt tuning work with GPT-5.

We've seen significant gains from applying these best practices and adopting our canonical tools whenever possible, and we hope that this guide, along with the prompt optimizer tool we've built, will serve as a launchpad for your use of GPT-5. But, as always, remember that prompting is not a one-size-fits-all exercise - we encourage you to run experiments and iterate on the foundation offered here to find the best solution for your problem.

Agentic workflow predictability

We trained GPT-5 with developers in mind: we've focused on improving tool calling, instruction following, and long-context understanding to serve as the best foundation model for agentic applications. If adopting GPT-5 for agentic and tool calling flows, we recommend upgrading to the Responses API, where reasoning is persisted between tool calls, leading to more efficient and intelligent outputs.

Controlling agentic eagerness

Agentic scaffolds can span a wide spectrum of control—some systems delegate the vast majority of decision-making to the underlying model, while others keep the model on a tight leash with heavy programmatic logical branching. GPT-5 is trained to operate anywhere along this spectrum, from making high-level decisions under ambiguous circumstances to handling focused, well-defined tasks. In this section we cover how to best calibrate GPT-5's agentic eagerness: in other words, its balance between proactivity and awaiting explicit guidance.

Prompting for less eagerness

GPT-5 is, by default, thorough and comprehensive when trying to gather context in an agentic environment to ensure it will produce a correct answer. To reduce the scope of GPT-5's agentic behavior—including limiting tangential tool-calling action and minimizing latency to reach a final answer—try the following:

- Switch to a lower `reasoning_effort`. This reduces exploration depth but improves efficiency and latency. Many workflows can be accomplished with consistent results at medium or even low `reasoning_effort`.
- Define clear criteria in your prompt for how you want the model to explore the problem space. This reduces the model's need to explore and reason about too many ideas:

```
<context_gathering>
Goal: Get enough context fast. Parallelize discovery and stop as soon as you can act.

Method:
- Start broad, then fan out to focused subqueries.
- In parallel, launch varied queries; read top hits per query. Deduplicate paths and cache; don't repeat queries.
- Avoid over searching for context. If needed, run targeted searches in one parallel batch.

Early stop criteria:
- You can name exact content to change.
- Top hits converge (~70%) on one area/path.

Escalate once:
- If signals conflict or scope is fuzzy, run one refined parallel batch, then proceed.

Depth:
- Trace only symbols you'll modify or whose contracts you rely on; avoid transitive expansion unless necessary.

Loop:
- Batch search → minimal plan → complete task.
- Search again only if validation fails or new unknowns appear. Prefer acting over more searching.
</context_gathering>
```

If you're willing to be maximally prescriptive, you can even set fixed tool call budgets, like the one below. The budget can naturally vary based on your desired search depth.

```
<context_gathering>
- Search depth: very low
- Bias strongly towards providing a correct answer as quickly as possible, even if it might not be fully correct.
- Usually, this means an absolute maximum of 2 tool calls.
- If you think that you need more time to investigate, update the user with your latest findings and open questions. You can proceed if t
</context_gathering>
```

When limiting core context gathering behavior, it's helpful to explicitly provide the model with an escape hatch that makes it easier to satisfy a shorter context gathering step. Usually this comes in the form of a clause that allows the model to proceed under uncertainty, like "`even if it might not be fully correct`" in the above example.

Prompting for more eagerness

On the other hand, if you'd like to encourage model autonomy, increase tool-calling persistence, and reduce occurrences of clarifying questions or otherwise handing back to the user, we recommend increasing `reasoning_effort`, and using a prompt like the following to encourage persistence and thorough task completion:

```
<persistence>
- You are an agent - please keep going until the user's query is completely resolved, before ending your turn and yielding back to the us
```

```
- Only terminate your turn when you are sure that the problem is solved.
- Never stop or hand back to the user when you encounter uncertainty — research or deduce the most reasonable approach and continue.
- Do not ask the human to confirm or clarify assumptions, as you can always adjust later — decide what the most reasonable assumption is,
</persistence>
```

Generally, it can be helpful to clearly state the stop conditions of the agentic tasks, outline safe versus unsafe actions, and define when, if ever, it's acceptable for the model to hand back to the user. For example, in a set of tools for shopping, the checkout and payment tools should explicitly have a lower uncertainty threshold for requiring user clarification, while the search tool should have an extremely high threshold; likewise, in a coding setup, the delete file tool should have a much lower threshold than a grep search tool.

Tool preambles

We recognize that on agentic trajectories monitored by users, intermittent model updates on what it's doing with its tool calls and why can provide for a much better interactive user experience - the longer the rollout, the bigger the difference these updates make. To this end, GPT-5 is trained to provide clear upfront plans and consistent progress updates via "tool preamble" messages.

You can steer the frequency, style, and content of tool preambles in your prompt—from detailed explanations of every single tool call to a brief upfront plan and everything in between. This is an example of a high-quality preamble prompt:

```
<tool_preambles>
- Always begin by rephrasing the user's goal in a friendly, clear, and concise manner, before calling any tools.
- Then, immediately outline a structured plan detailing each logical step you'll follow. - As you execute your file edit(s), narrate each
- Finish by summarizing completed work distinctly from your upfront plan.
</tool_preambles>
```

Here's an example of a tool preamble that might be emitted in response to such a prompt—such preambles can drastically improve the user's ability to follow along with your agent's work as it grows more complicated:


```
"output": [
    {
      "id": "rs_6888f6d0606c819aa8205ecee386963f0e683233d39188e7",
      "type": "reasoning",
      "summary": [
        {
          "type": "summary_text",
          "text": "**Determining weather response**\n\nI need to answer the user's question about the weather in San Francisco. ...."
        },
    },
    {
      "id": "msg_6888f6d83acc819a978b51e772f0a5f40e683233d39188e7",
      "type": "message",
      "status": "completed",
      "content": [
        {
          "type": "output_text",
          "text": "I\u2019m going to check a live weather service to get the current conditions in San Francisco, providing the temperatu
        }
      ],
      "role": "assistant"
    },
    {
      "id": "fc_6888f6d86e28819aaaa1ba69cca766b70e683233d39188e7",
      "type": "function_call",
      "status": "completed",
      "arguments": "{\"location\":\"San Francisco, CA\",\"unit\":\"f\"}",
      "call_id": "call_XOnF4B9DvB8EJVB3JvWnGg83",
      "name": "get_weather"
    },
  ],
```


Reasoning effort

We provide a `reasoning_effort` parameter to control how hard the model thinks and how willingly it calls tools; the default is `medium`, but you should scale up or down depending on the difficulty of your task. For complex, multi-step tasks, we recommend higher reasoning to ensure the best possible outputs. Moreover, we observe peak performance when distinct, separable tasks are broken up across multiple agent turns, with one turn for each task.

Reusing reasoning context with the Responses API

We strongly recommend using the Responses API when using GPT-5 to unlock improved agentic flows, lower costs, and more efficient token usage in your applications.

We've seen statistically significant improvements in evaluations when using the Responses API over Chat Completions—for example, we observed Tau-Bench Retail score increases from 73.9% to 78.2% just by switching to the Responses API and including `previous_response_id` to pass back previous reasoning items into subsequent requests. This allows the model to refer to its previous reasoning traces, conserving CoT tokens and eliminating the need to reconstruct a plan from scratch after each tool call, improving both latency and performance - this feature is available for all Responses API users, including ZDR organizations.

Maximizing coding performance, from planning to execution

GPT-5 leads all frontier models in coding capabilities: it can work in large codebases to fix bugs, handle large diffs, and implement multi-file refactors or large new features. It also excels at implementing new apps entirely from scratch, covering both frontend and backend implementation. In this section, we'll discuss prompt optimizations that we've seen improve programming performance in production use cases for our coding agent customers.

Frontend app development

GPT-5 is trained to have excellent baseline aesthetic taste alongside its rigorous implementation abilities. We're confident in its ability to use all types of web development frameworks and packages; however, for new apps, we recommend using the following frameworks and packages to get the most out of the model's frontend capabilities:

- Frameworks: Next.js (TypeScript), React, HTML
- Styling / UI: Tailwind CSS, shadcn/ui, Radix Themes
- Icons: Material Symbols, Heroicons, Lucide
- Animation: Motion
- Fonts: San Serif, Inter, Geist, Mona Sans, IBM Plex Sans, Manrope

Zero-to-one app generation

GPT-5 is excellent at building applications in one shot. In early experimentation with the model, users have found that prompts like the one below—asking the model to iteratively execute against self-constructed excellence rubrics—improve output quality by using GPT-5's thorough planning and self-reflection capabilities.

```
<self_reflection>
- First, spend time thinking of a rubric until you are confident.
- Then, think deeply about every aspect of what makes for a world-class one-shot web app. Use that knowledge to create a rubric that has
- Finally, use the rubric to internally think and iterate on the best possible solution to the prompt that is provided. Remember that if
</self_reflection>
```

Matching codebase design standards

When implementing incremental changes and refactors in existing apps, model-written code should adhere to existing style and design standards, and "blend in" to the codebase as neatly as possible. Without special prompting, GPT-5 already searches for reference context from the codebase - for example reading package.json to view already installed packages - but this behavior can be further enhanced with prompt directions that summarize key aspects like engineering principles, directory structure, and best practices of the codebase, both explicit and implicit. The prompt snippet below demonstrates one way of organizing code editing rules for GPT-5: feel free to change the actual content of the rules according to your programming design taste!

```
<code_editing_rules>
<guiding_principles>
- Clarity and Reuse: Every component and page should be modular and reusable. Avoid duplication by factoring repeated UI patterns into co
- Consistency: The user interface must adhere to a consistent design system—color tokens, typography, spacing, and components must be uni
- Simplicity: Favor small, focused components and avoid unnecessary complexity in styling or logic.
- Demo-Oriented: The structure should allow for quick prototyping, showcasing features like streaming, multi-turn conversations, and tool
- Visual Quality: Follow the high visual quality bar as outlined in OSS guidelines (spacing, padding, hover states, etc.)
</guiding_principles>

<frontend_stack_defaults>
- Framework: Next.js (TypeScript)
- Styling: TailwindCSS
- UI Components: shadcn/ui
- Icons: Lucide
- State Management: Zustand
- Directory Structure:
\`\`\`
/src
 /app
   /api/<route>/route.ts         # API endpoints
   /(pages)                      # Page routes
 /components/                    # UI building blocks
 /hooks/                         # Reusable React hooks
 /lib/                           # Utilities (fetchers, helpers)
 /stores/                        # Zustand stores
 /types/                         # Shared TypeScript types
 /styles/                        # Tailwind config
\`\`\`
</frontend_stack_defaults>

<ui_ux_best_practices>
- Visual Hierarchy: Limit typography to 4–5 font sizes and weights for consistent hierarchy; use `text-xs` for captions and annotations;
- Color Usage: Use 1 neutral base (e.g., `zinc`) and up to 2 accent colors.
- Spacing and Layout: Always use multiples of 4 for padding and margins to maintain visual rhythm. Use fixed height containers with inter
- State Handling: Use skeleton placeholders or `animate-pulse` to indicate data fetching. Indicate clickability with hover transitions (`
- Accessibility: Use semantic HTML and ARIA roles where appropriate. Favor pre-built Radix/shadcn components, which have accessibility ba
</ui_ux_best_practices>

<code_editing_rules>
```

Collaborative coding in production: Cursor's GPT-5 prompt tuning

We're proud to have had AI code editor Cursor as a trusted alpha tester for GPT-5: below, we show a peek into how Cursor tuned their prompts to get the most out of the model's capabilities. For more information, their team has also published a blog post detailing GPT-5's day-one integration into Cursor: https://cursor.com/blog/gpt-5

System prompt and parameter tuning

Cursor's system prompt focuses on reliable tool calling, balancing verbosity and autonomous behavior while giving users the ability to configure custom instructions. Cursor's goal for their system prompt is to allow the Agent to operate relatively autonomously during long horizon tasks, while still faithfully following user-provided instructions.

The team initially found that the model produced verbose outputs, often including status updates and post-task summaries that, while technically relevant, disrupted the natural flow of the user; at the same time, the code outputted in tool calls was high quality, but sometimes hard to read due to terseness, with single-letter variable names dominant. In search of a better balance, they set the verbosity API parameter to low to keep text outputs brief, and then modified the prompt to strongly encourage verbose outputs in coding tools only.

```
Write code for clarity first. Prefer readable, maintainable solutions with clear names, comments where needed, and straightforward contro
```

This dual usage of parameter and prompt resulted in a balanced format combining efficient, concise status updates and final work summary with much more readable code diffs.

Cursor also found that the model occasionally deferred to the user for clarification or next steps before taking action, which created unnecessary friction in the flow of longer tasks. To address this, they found that including not just available tools and surrounding context, but also more details about product behavior encouraged the model to carry out longer tasks with minimal interruption and greater autonomy. Highlighting specifics of Cursor features such as Undo/Reject code and user preferences helped reduce ambiguity by clearly specifying how GPT-5 should behave in its environment. For longer horizon tasks, they found this prompt improved performance:

```
Be aware that the code edits you make will be displayed to the user as proposed changes, which means (a) your code edits can be quite pro
```

Cursor found that sections of their prompt that had been effective with earlier models needed tuning to get the most out of GPT-5. Here is one example below:

```
<maximize_context_understanding>
Be THOROUGH when gathering information. Make sure you have the FULL picture before replying. Use additional tool calls or clarifying ques
...
</maximize_context_understanding>
```

While this worked well with older models that needed encouragement to analyze context thoroughly, they found it counterproductive with GPT-5, which is already naturally introspective and proactive at gathering context. On smaller tasks, this prompt often caused the model to overuse tools by calling search repetitively, when internal knowledge would have been sufficient.

To solve this, they refined the prompt by removing the maximize_ prefix and softening the language around thoroughness. With this adjusted instruction in place, the Cursor team saw GPT-5 make better decisions about when to rely on internal knowledge versus reaching for external tools. It maintained a high level of autonomy without unnecessary tool usage, leading to more efficient and relevant behavior. In Cursor's testing, using structured XML specs like <[instruction]_spec> improved instruction adherence on their prompts and allows them to clearly reference previous categories and sections elsewhere in their prompt.

```
<context_understanding>
...
If you've performed an edit that may partially fulfill the USER's query, but you're not confident, gather more information or use more to
Bias towards not asking the user for help if you can find the answer yourself.
</context_understanding>
```

While the system prompt provides a strong default foundation, the user prompt remains a highly effective lever for steerability. GPT-5 responds well to direct and explicit instruction and the Cursor team has consistently seen that structured, scoped prompts yield the most reliable results. This includes areas like verbosity control, subjective code style preferences, and sensitivity to edge cases. Cursor found allowing users to configure their own custom Cursor rules to be particularly impactful with GPT-5's improved steerability, giving their users a more customized experience.

Optimizing intelligence and instruction-following

Steering

As our most steerable model yet, GPT-5 is extraordinarily receptive to prompt instructions surrounding verbosity, tone, and tool calling behavior.

Verbosity

In addition to being able to control the reasoning_effort as in previous reasoning models, in GPT-5 we introduce a new API parameter called verbosity, which influences the length of the model's final answer, as opposed to the length of its thinking. Our blog post covers the idea behind this parameter in more detail - but in this guide, we'd like to emphasize that while the API verbosity parameter is the default for the rollout, GPT-5 is trained to respond to natural-language verbosity overrides in the prompt for specific contexts where you might want the model to deviate from the global default. Cursor's example above of setting low verbosity globally, and then specifying high verbosity only for coding tools, is a prime example of such a context.

Instruction following

Like GPT-4.1, GPT-5 follows prompt instructions with surgical precision, which enables its flexibility to drop into all types of workflows. However, its careful instruction-following behavior means that poorly-constructed prompts containing contradictory or vague instructions can be more damaging to GPT-5 than to other models, as it expends reasoning tokens searching for a way to reconcile the contradictions rather than picking one instruction at random.

Below, we give an adversarial example of the type of prompt that often impairs GPT-5's reasoning traces - while it may appear internally consistent at first glance, a closer inspection reveals conflicting instructions regarding appointment scheduling:

- `Never schedule an appointment without explicit patient consent recorded in the chart` conflicts with the subsequent `auto-assign the earliest same-day slot without contacting the patient as the first action to reduce risk.`
- The prompt says `Always look up the patient profile before taking any other actions to ensure they are an existing patient.` but then continues with the contradictory instruction `When symptoms indicate high urgency, escalate as EMERGENCY and direct the patient to call 911 immediately before any scheduling step.`

```
You are CareFlow Assistant, a virtual admin for a healthcare startup that schedules patients based on priority and symptoms. Your goal is

- Core entities include Patient, Provider, Appointment, and PriorityLevel (Red, Orange, Yellow, Green). Map symptoms to priority: Red wit
+Core entities include Patient, Provider, Appointment, and PriorityLevel (Red, Orange, Yellow, Green). Map symptoms to priority: Red with
*Do not do lookup in the emergency case, proceed immediately to providing 911 guidance.*

- Use the following capabilities: schedule-appointment, modify-appointment, waitlist-add, find-provider, lookup-patient and notify-patien

- For high-acuity Red and Orange cases, auto-assign the earliest same-day slot *without contacting* the patient *as the first action to r

- For high-acuity Red and Orange cases, auto-assign the earliest same-day slot *after informing* the patient *of your actions.* If a suit
```

By resolving the instruction hierarchy conflicts, GPT-5 elicits much more efficient and performant reasoning. We fixed the contradictions by:

- Changing auto-assignment to occur after contacting a patient, auto-assign the earliest same-day slot after informing the patient of your actions. to be consistent with only scheduling with consent.
- Adding Do not do lookup in the emergency case, proceed immediately to providing 911 guidance. to let the model know it is ok to not look up in case of emergency.

We understand that the process of building prompts is an iterative one, and many prompts are living documents constantly being updated by different stakeholders - but this is all the more reason to thoroughly review them for poorly-worded instructions. Already, we've seen multiple early users uncover ambiguities and contradictions in their core prompt libraries upon conducting such a review: removing them drastically streamlined and improved their GPT-5 performance. We recommend testing your prompts in our prompt optimizer tool to help identify these types of issues.

Minimal reasoning

In GPT-5, we introduce minimal reasoning effort for the first time: our fastest option that still reaps the benefits of the reasoning model paradigm. We consider this to be the best upgrade for latency-sensitive users, as well as current users of GPT-4.1.

Perhaps unsurprisingly, we recommend prompting patterns that are similar to GPT-4.1 for best results. minimal reasoning performance can vary more drastically depending on prompt than higher reasoning levels, so key points to emphasize include:

1. Prompting the model to give a brief explanation summarizing its thought process at the start of the final answer, for example via a bullet point list, improves performance on tasks requiring higher intelligence.
2. Requesting thorough and descriptive tool-calling preambles that continually update the user on task progress improves performance in agentic workflows.
3. Disambiguating tool instructions to the maximum extent possible and inserting agentic persistence reminders as shared above, are particularly critical at minimal reasoning to maximize agentic ability in long-running rollout and prevent premature termination.
4. Prompted planning is likewise more important, as the model has fewer reasoning tokens to do internal planning. Below, you can find a sample planning prompt snippet we placed at the beginning of an agentic task: the second paragraph especially ensures that the agent fully completes the task and all subtasks before yielding back to the user.

```
Remember, you are an agent - please keep going until the user's query is completely resolved, before ending your turn and yielding back t

You must plan extensively in accordance with the workflow steps before making subsequent function calls, and reflect extensively on the o
```

Markdown formatting

By default, GPT-5 in the API does not format its final answers in Markdown, in order to preserve maximum compatibility with developers whose applications may not support Markdown rendering. However, prompts like the following are largely successful in inducing hierarchical Markdown final answers.

```
- Use Markdown **only where semantically correct** (e.g., `inline code`, ```code fences```, lists, tables).
- When using markdown in assistant messages, use backticks to format file, directory, function, and class names. Use \( and \) for inline
```

Occasionally, adherence to Markdown instructions specified in the system prompt can degrade over the course of a long conversation. In the event that you experience this, we've seen consistent adherence from appending a Markdown instruction every 3-5 user messages.

Metaprompting

Finally, to close with a meta-point, early testers have found great success using GPT-5 as a meta-prompter for itself. Already, several users have deployed prompt revisions to production that were generated simply by asking GPT-5 what elements could be added to an unsuccessful prompt to elicit a desired behavior, or removed to prevent an undesired one.

Here is an example metaprompt template we liked:

```
When asked to optimize prompts, give answers from your own perspective - explain what specific phrases could be added to, or deleted from

Here's a prompt: [PROMPT]

The desired behavior from this prompt is for the agent to [DO DESIRED BEHAVIOR], but instead it [DOES UNDESIRED BEHAVIOR]. While keeping
```

Appendix

SWE-Bench verified developer instructions

```
In this environment, you can run `bash -lc <apply_patch_command>` to execute a diff/patch against a file, where <apply_patch_command> is

apply_patch << 'PATCH'
*** Begin Patch
[YOUR_PATCH]
*** End Patch
PATCH

Where [YOUR_PATCH] is the actual content of your patch.

Always verify your changes extremely thoroughly. You can make as many tool calls as you like - the user is very patient and prioritizes c
IMPORTANT: not all tests are visible to you in the repository, so even on problems you think are relatively straightforward, you must dou
```

Agentic coding tool definitions

```
## Set 1: 4 functions, no terminal

type apply_patch = (_: {
patch: string, // default: null
}) => any;

type read_file = (_: {
path: string, // default: null
```

```
line_start?: number, // default: 1
line_end?: number, // default: 20
}) => any;

type list_files = (_: {
path?: string, // default: ""
depth?: number, // default: 1
}) => any;

type find_matches = (_: {
query: string, // default: null
path?: string, // default: ""
max_results?: number, // default: 50
}) => any;

## Set 2: 2 functions, terminal-native

type run = (_: {
command: string[], // default: null
session_id?: string | null, // default: null
working_dir?: string | null, // default: null
ms_timeout?: number | null, // default: null
environment?: object | null, // default: null
run_as_user?: string | null, // default: null
}) => any;

type send_input = (_: {
session_id: string, // default: null
text: string, // default: null
wait_ms?: number, // default: 100
}) => any;
```

As shared in the GPT-4.1 prompting guide, here is our most updated `apply_patch` implementation: we highly recommend using `apply_patch` for file edits to match the training distribution. The newest implementation should match the GPT-4.1 implementation in the overwhelming majority of cases.

Taubench-Retail minimal reasoning instructions

```
As a retail agent, you can help users cancel or modify pending orders, return or exchange delivered orders, modify their default user add

Remember, you are an agent - please keep going until the user's query is completely resolved, before ending your turn and yielding back t

If you are not sure about information pertaining to the user's request, use your tools to read files and gather the relevant information:

You MUST plan extensively before each function call, and reflect extensively on the outcomes of the previous function calls, ensuring use

# Workflow steps
- At the beginning of the conversation, you have to authenticate the user identity by locating their user id via email, or via name + zip
- Once the user has been authenticated, you can provide the user with information about order, product, profile information, e.g. help th
- You can only help one user per conversation (but you can handle multiple requests from the same user), and must deny any requests for t
- Before taking consequential actions that update the database (cancel, modify, return, exchange), you have to list the action detail and
- You should not make up any information or knowledge or procedures not provided from the user or the tools, or give subjective recommend
- You should at most make one tool call at a time, and if you take a tool call, you should not respond to the user at the same time. If y
- You should transfer the user to a human agent if and only if the request cannot be handled within the scope of your actions.

## Domain basics
- All times in the database are EST and 24 hour based. For example "02:30:00" means 2:30 AM EST.
- Each user has a profile of its email, default address, user id, and payment methods. Each payment method is either a gift card, a paypa
- Our retail store has 50 types of products. For each type of product, there are variant items of different options. For example, for a '
- Each product has an unique product id, and each item has an unique item id. They have no relations and should not be confused.
- Each order can be in status 'pending', 'processed', 'delivered', or 'cancelled'. Generally, you can only take action on pending or deli
- Exchange or modify order tools can only be called once. Be sure that all items to be changed are collected into a list before making th

## Cancel pending order
- An order can only be cancelled if its status is 'pending', and you should check its status before taking the action.
- The user needs to confirm the order id and the reason (either 'no longer needed' or 'ordered by mistake') for cancellation.
- After user confirmation, the order status will be changed to 'cancelled', and the total will be refunded via the original payment metho

## Modify pending order
- An order can only be modified if its status is 'pending', and you should check its status before taking the action.
- For a pending order, you can take actions to modify its shipping address, payment method, or product item options, but nothing else.

## Modify payment
- The user can only choose a single payment method different from the original payment method.
- If the user wants the modify the payment method to gift card, it must have enough balance to cover the total amount.
- After user confirmation, the order status will be kept 'pending'. The original payment method will be refunded immediately if it is a g

## Modify items
- This action can only be called once, and will change the order status to 'pending (items modifed)', and the agent will not be able to m
- For a pending order, each item can be modified to an available new item of the same product but of different product option. There cann
- The user must provide a payment method to pay or receive refund of the price difference. If the user provides a gift card, it must have

## Return delivered order
- An order can only be returned if its status is 'delivered', and you should check its status before taking the action.
- The user needs to confirm the order id, the list of items to be returned, and a payment method to receive the refund.
- The refund must either go to the original payment method, or an existing gift card.
- After user confirmation, the order status will be changed to 'return requested', and the user will receive an email regarding how to re

## Exchange delivered order
- An order can only be exchanged if its status is 'delivered', and you should check its status before taking the action. In particular, r
- For a delivered order, each item can be exchanged to an available new item of the same product but of different product option. There c
- The user must provide a payment method to pay or receive refund of the price difference. If the user provides a gift card, it must have
- After user confirmation, the order status will be changed to 'exchange requested', and the user will receive an email regarding how to
```

Terminal-Bench prompt

Please resolve the user's task by editing and testing the code files in your current code execution session.
You are a deployed coding agent.
Your session is backed by a container specifically designed for you to easily modify and run code.
You MUST adhere to the following criteria when executing the task:

<instructions>
- Working on the repo(s) in the current environment is allowed, even if they are proprietary.
- Analyzing code for vulnerabilities is allowed.
- Showing user code and tool call details is allowed.
- User instructions may overwrite the _CODING GUIDELINES_ section in this developer message.
- Do not use \`ls -R\`, \`find\`, or \`grep\` - these are slow in large repos. Use \`rg\` and \`rg --files\`.
- Use \`apply_patch\` to edit files: {"cmd":["apply_patch","*** Begin Patch\\n*** Update File: path/to/file.py\\n@@ def example():\\n- pa
- If completing the user's task requires writing or modifying files:
 - Your code and final answer should follow these _CODING GUIDELINES_:
   - Fix the problem at the root cause rather than applying surface-level patches, when possible.
   - Avoid unneeded complexity in your solution.
     - Ignore unrelated bugs or broken tests; it is not your responsibility to fix them.
   - Update documentation as necessary.
   - Keep changes consistent with the style of the existing codebase. Changes should be minimal and focused on the task.
     - Use \`git log\` and \`git blame\` to search the history of the codebase if additional context is required; internet access is disa
   - NEVER add copyright or license headers unless specifically requested.
   - You do not need to \`git commit\` your changes; this will be done automatically for you.
   - If there is a .pre-commit-config.yaml, use \`pre-commit run --files ...\` to check that your changes pass the pre- commit checks. Ho
     - If pre-commit doesn't work after a few retries, politely inform the user that the pre-commit setup is broken.
   - Once you finish coding, you must
     - Check \`git status\` to sanity check your changes; revert any scratch files or changes.
     - Remove all inline comments you added much as possible, even if they look normal. Check using \`git diff\`. Inline comments must be
     - Check if you accidentally add copyright or license headers. If so, remove them.
     - Try to run pre-commit if it is available.
     - For smaller tasks, describe in brief bullet points
     - For more complex tasks, include brief high-level description, use bullet points, and include details that would be relevant to a c
- If completing the user's task DOES NOT require writing or modifying files (e.g., the user asks a question about the code base):
 - Respond in a friendly tune as a remote teammate, who is knowledgeable, capable and eager to help with coding.
- When your task involves writing or modifying files:
 - Do NOT tell the user to "save the file" or "copy the code into a file" if you already created or modified the file using \`apply_patch
 - Do NOT show the full contents of large files you have already written, unless the user explicitly asks for them.
</instructions>

<apply_patch>
To edit files, ALWAYS use the \`shell\` tool with \`apply_patch\` CLI. \`apply_patch\` effectively allows you to execute a diff/patch ag
\`\`\`bash
{"cmd": ["apply_patch", "<<'EOF'\\n*** Begin Patch\\n[YOUR_PATCH]\\n*** End Patch\\nEOF\\n"], "workdir": "..."}
\`\`\`
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 previo
- If 3 lines of context is insufficient to uniquely identify the snippet of code within the file, use the @@ operator to indicate the cla
@@ 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
@@ 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 mess
\`\`\`bash
{"cmd": ["apply_patch", "<<'EOF'\\n*** Begin Patch\\n*** Update File: pygorithm/searching/binary_search.py\\n@@ class BaseClass\\n@@
\`\`\`
File references can only be relative, NEVER ABSOLUTE. After the apply_patch command is run, it will always say "Done!", regardless of whe
</apply_patch>

<persistence>
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
- Never stop at uncertainty - research or deduce the most reasonable approach and continue.
- Do not ask the human to confirm assumptions - document them, act on them, and adjust mid-task if proven wrong.
</persistence>

<exploration>
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 th
Before coding, always:
- Decompose the request into explicit requirements, unclear areas, and hidden assumptions.
- Map the scope: identify the codebase regions, files, functions, or libraries likely involved. If unknown, plan and perform targeted sea
- Check dependencies: identify relevant frameworks, APIs, config files, data formats, and versioning concerns.
- Resolve ambiguity proactively: choose the most probable interpretation based on repo context, conventions, and dependency docs.
- Define the output contract: exact deliverables such as files changed, expected outputs, API responses, CLI behavior, and tests passing.
- Formulate an execution plan: research steps, implementation sequence, and testing strategy in your own words and refer to it as you wor
</exploration>

<verification>
Routinely verify your code works as you work through the task, especially any deliverables to ensure they run properly. Don't hand back t
Exit excessively long running processes and optimize your code to run faster.
</verification>

<efficiency>
Efficiency is key. you have a time limit. Be meticulous in your planning, tool calling, and verification so you don't waste time.
</efficiency>

<final_instructions>
Never use editor tools to edit files. Always use the \`apply_patch\` tool.
</final_instructions>