


# Reasoning models

Copy page

Explore advanced reasoning and problem-solving models.

**Reasoning models** like o3 and o4-mini are LLMs trained with reinforcement learning to perform reasoning. Reasoning models **think before they answer**, producing a long internal chain of thought before responding to the user. Reasoning models excel in complex problem solving, coding, scientific reasoning, and multi-step planning for agentic workflows. They're also the best models for **Codex CLI**, our lightweight coding agent.

As with our GPT series, we provide smaller, faster models ( o4-mini and o3-mini ) that are less expensive per token. The larger models ( o3 and o1 ) are slower and more expensive but often generate better responses for complex tasks and broad domains.

 To ensure safe deployment of our latest reasoning models o3 and o4-mini , some developers may need to complete organization verification before accessing these models. Get started with verification on the platform settings page.

## Get started with reasoning

Reasoning models can be used through the Responses API as seen here.

Using a reasoning model in the Responses APIpython

```
1  from openai import OpenAI
2
3  client = OpenAI()
4
5  prompt = """
6  Write a bash script that takes a matrix represented as a string with
7  format '[1,2],[3,4],[5,6]' and prints the transpose in the same format.
8  """
9
10 response = client.responses.create(
11     model="o4-mini",
12     reasoning={"effort": "medium"},
13     input=[
14         {
15             "role": "user",
16             "content": prompt
17         }
18     ]
19 )
20
21 print(response.output_text)
```

In the example above, the reasoning.effort parameter guides the model on how many reasoning tokens to generate before creating a response to the prompt.

Specify low , medium , or high for this parameter, where low favors speed and economical token usage, and high favors more complete reasoning. The default value is medium , which is a balance between speed and reasoning accuracy.

## How reasoning works

Reasoning models introduce **reasoning tokens** in addition to input and output tokens. The models use these reasoning tokens to "think," breaking down the prompt and considering multiple approaches to generating a response. After generating reasoning tokens, the model produces an answer as visible completion tokens and discards the reasoning tokens from its context.

Responses

Get started

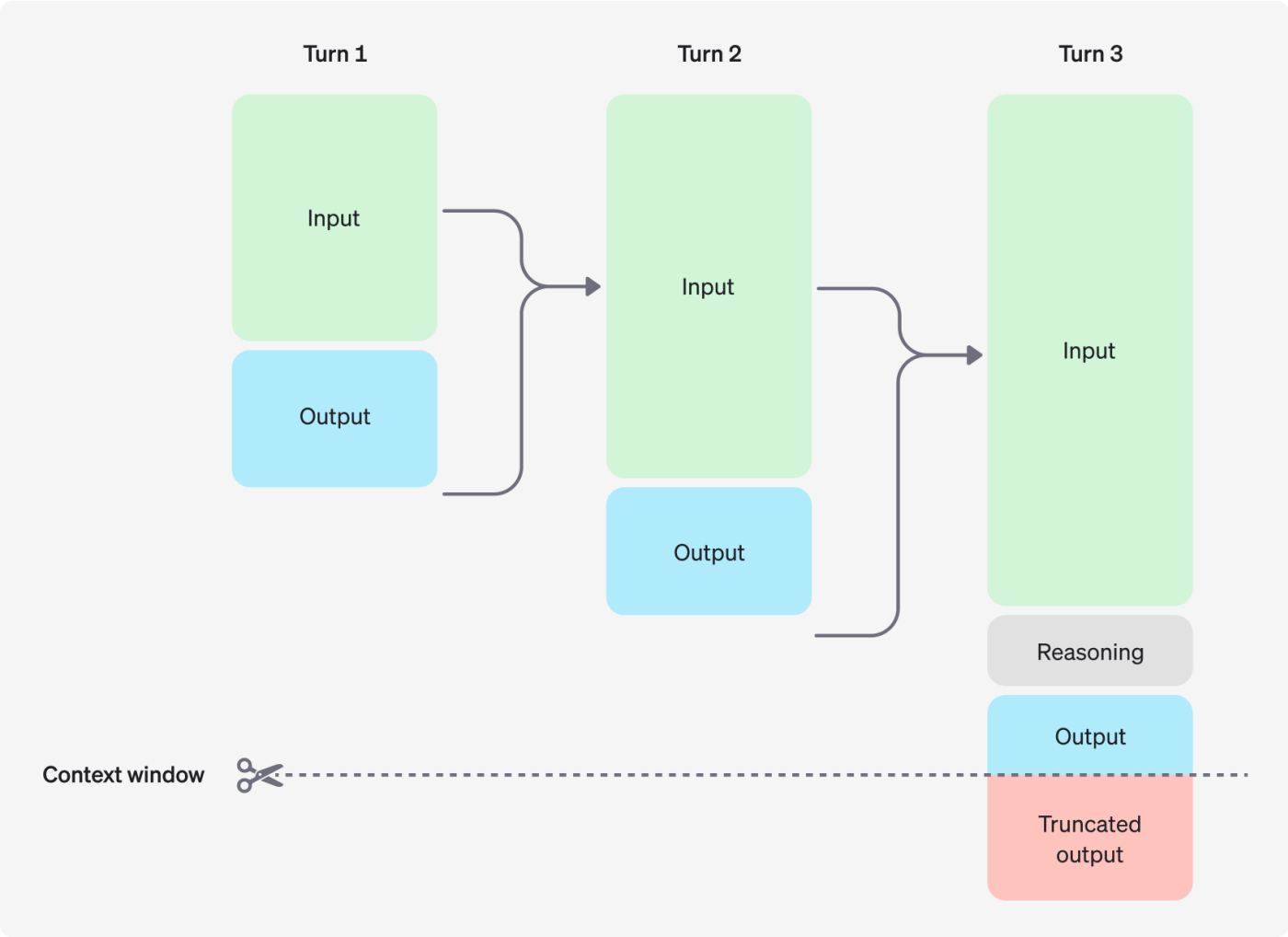
How reasoning works

Reasoning summaries

Advice on prompting

Use case examples

Here is an example of a multi-step conversation between a user and an assistant. Input and output tokens from each step are carried over, while reasoning tokens are discarded.



While reasoning tokens are not visible via the API, they still occupy space in the model's context window and are billed as **output tokens**.

### Managing the context window

It's important to ensure there's enough space in the context window for reasoning tokens when creating responses. Depending on the problem's complexity, the models may generate anywhere from a few hundred to tens of thousands of reasoning tokens. The exact number of reasoning tokens used is visible in the **usage object of the response object**, under `output_tokens_details` :

```
1 {
2   "usage": {
3     "input_tokens": 75,
4     "input_tokens_details": {
5       "cached_tokens": 0
6     },
7     "output_tokens": 1186,
8     "output_tokens_details": {
9       "reasoning_tokens": 1024
10    },
11    "total_tokens": 1261
12  }
13 }
```

Context window lengths are found on the [model reference page](#), and will differ across model snapshots.

### Controlling costs

If you're managing context manually across model turns, you can discard older reasoning items *unless* you're responding to a function call, in which case you must include all reasoning items between the function call and the last user message.

To manage costs with reasoning models, you can limit the total number of tokens the model generates (including both reasoning and final output tokens) by using the `max_output_tokens` parameter.

## Allocating space for reasoning

If the generated tokens reach the context window limit or the `max_output_tokens` value you've set, you'll receive a response with a `status` of `incomplete` and `incomplete_details` with `reason` set to `max_output_tokens`. This might occur before any visible output tokens are produced, meaning you could incur costs for input and reasoning tokens without receiving a visible response.

To prevent this, ensure there's sufficient space in the context window or adjust the `max_output_tokens` value to a higher number. OpenAI recommends reserving at least 25,000 tokens for reasoning and outputs when you start experimenting with these models. As you become familiar with the number of reasoning tokens your prompts require, you can adjust this buffer accordingly.

Handling incomplete responses

python ↕

```
1  from openai import OpenAI
2
3  client = OpenAI()
4
5  prompt = """
6  Write a bash script that takes a matrix represented as a string with
7  format '[1,2],[3,4],[5,6]' and prints the transpose in the same format.
8  """
9
10 response = client.responses.create(
11     model="o4-mini",
12     reasoning={"effort": "medium"},
13     input=[
14         {
15             "role": "user",
16             "content": prompt
17         }
18     ],
19     max_output_tokens=300,
20 )
21
22
23 if response.status == "incomplete" and response.incomplete_details.reason == "max_out
24     print("Ran out of tokens")
25     if response.output_text:
26         print("Partial output:", response.output_text)
27     else:
28         print("Ran out of tokens during reasoning")
```

## Keeping reasoning items in context

When doing [function calling](#) with a reasoning model in the [Responses API](#), we highly recommend you pass back any reasoning items returned with the last function call (in addition to the output of your function). If the model calls multiple functions consecutively, you should pass back all reasoning items, function call items, and function call output items, since the last `user` message. This allows the model to continue its reasoning process to produce better results in the most token-efficient manner.

The simplest way to do this is to pass in all reasoning items from a previous response into the next one. Our systems will smartly ignore any reasoning items that aren't relevant to your functions, and only retain those in context that are relevant. You can pass reasoning items from previous responses either using the `previous_response_id` parameter, or by manually passing in all the [output](#) items from a past response into the [input](#) of a new one.

For advanced use-cases where you might be truncating and optimizing parts of the context window before passing them on to the next response, just ensure all items between the last user message and your function call output are passed into the next response untouched. This will ensure that the model has all the context it needs.

Check out [this guide](#) to learn more about manual context management.

## Reasoning summaries

While we don't expose the raw reasoning tokens emitted by the model, you can view a summary of the model's reasoning using the `summary` parameter.

Different models support different reasoning summarizers—for example, our computer use model supports the `concise` summarizer, while o4-mini supports `detailed`. To simply access the most detailed summarizer available, set the value of this parameter to `auto` and view the reasoning summary as part of the `summary` array in the `reasoning` `output` item.

This feature is also supported with streaming, and across the following reasoning models:

`o4-mini`, `o3`, `o3-mini` and `o1`.

Before using summarizers with our latest reasoning models, you may need to complete [organization verification](#) to ensure safe deployment. Get started with verification on the [platform settings page](#).

Generate a summary of the reasoning

```
1 reasoning: {
2   effort: "medium", // unchanged
3   summary: "auto" // auto gives you the best available summary (detailed > auto > None
4 }
```

## Advice on prompting

There are some differences to consider when prompting a reasoning model. Reasoning models provide better results on tasks with only high-level guidance, while GPT models often benefit from very precise instructions.

- A reasoning model is like a senior co-worker—you can give them a goal to achieve and trust them to work out the details.
- A GPT model is like a junior coworker—they'll perform best with explicit instructions to create a specific output.

For more information on best practices when using reasoning models, [refer to this guide](#).

## Prompt examples

Coding (refactoring) Coding (planning) STEM Research

OpenAI o-series models are able to implement complex algorithms and produce code. This prompt asks o1 to refactor a React component based on some specific criteria.


Refactor code javascript


```
1 import OpenAI from "openai";
2
3 const openai = new OpenAI();
4
5 const prompt = `
6 Instructions:
7 - Given the React component below, change it so that nonfiction books have red
8   text.
9 - Return only the code in your reply
10 - Do not include any additional formatting, such as markdown code blocks
11 - For formatting, use four space tabs, and do not allow any lines of code to
12   exceed 80 columns
13
14 const books = [
15   { title: 'Dune', category: 'fiction', id: 1 },
16   { title: 'Frankenstein', category: 'fiction', id: 2 },
17   { title: 'Moneyball', category: 'nonfiction', id: 3 },
18 ];
```

```
20 export default function BookList() {
21   const listItems = books.map(book =>
22     <li>
23       {book.title}
24     </li>
25   );
26
27   return (
28     <ul>{listItems}</ul>
29   );
30 }
31 `.trim();
32
33
34 const response = await openai.responses.create({
35   model: "o4-mini",
36   input: [
37     {
38       role: "user",
39       content: prompt,
40     },
41   ],
42 });
43
44 console.log(response.output_text);
```

## Use case examples

Some examples of using reasoning models for real-world use cases can be found in [the cookbook](#).

- 

**Using reasoning for data validation**  
Evaluate a synthetic medical data set for discrepancies.
- 

**Using reasoning for routine generation**  
Use help center articles to generate actions that an agent could perform.