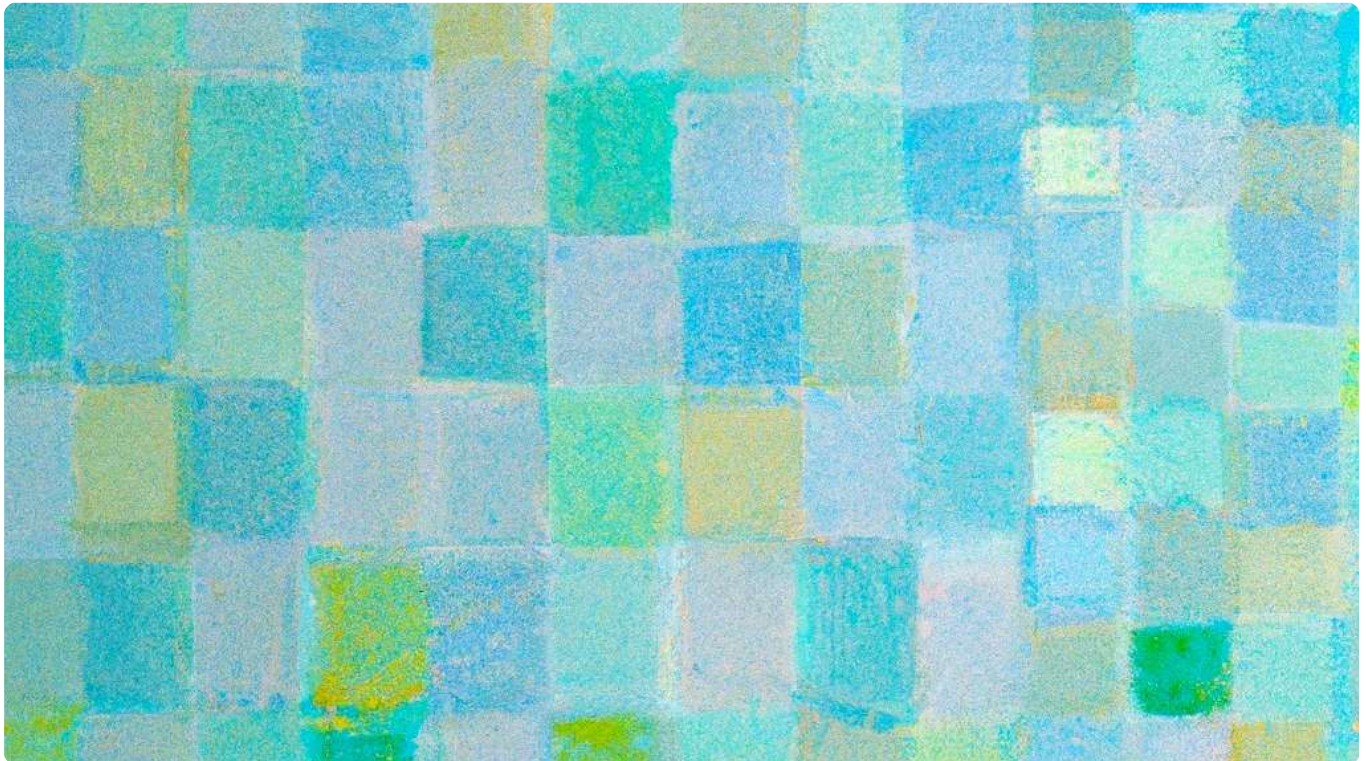





August 6, 2024 Company

Introducing Structured Outputs in the API

We are introducing Structured Outputs in the API—model outputs now reliably adhere to developer-supplied JSON Schemas.



 Share



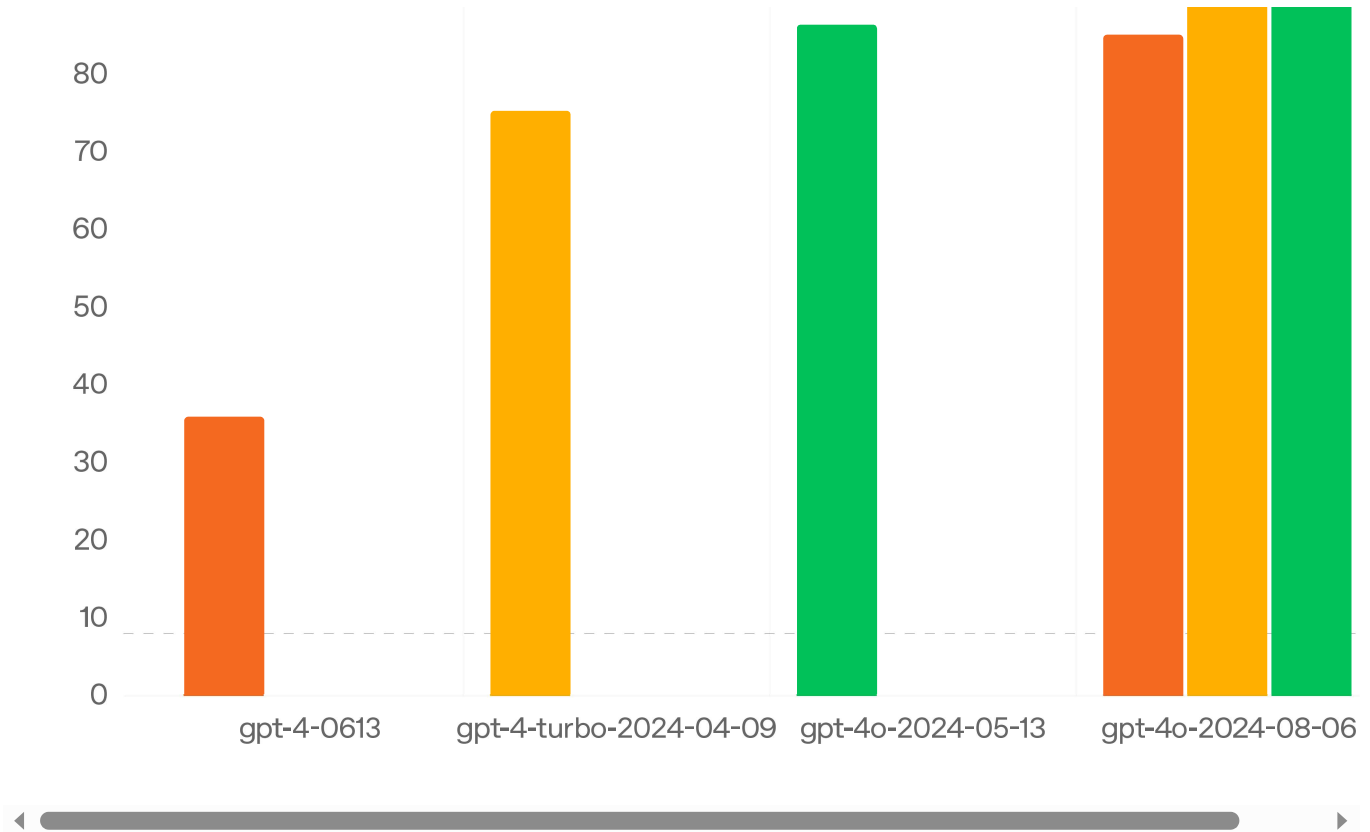
looking to build reliable applications with our models. While JSON mode improves model reliability for generating valid JSON outputs, it does not guarantee that the model's response will conform to a particular schema. Today we're introducing Structured Outputs in the API, a new feature designed to ensure model-generated outputs will exactly match JSON Schemas provided by developers.

Generating structured data from unstructured inputs is one of the core use cases for AI in today's applications. Developers use the OpenAI API to build powerful assistants that have the ability to fetch data and answer questions via function calling, extract structured data for data entry, and build multi-step agentic workflows that allow LLMs to take actions. Developers have long been working around the limitations of LLMs in this area via open source tooling, prompting, and retrying requests repeatedly to ensure that model outputs match the formats needed to interoperate with their systems. Structured Outputs solves this problem by constraining OpenAI models to match developer-supplied schemas and by training our models to better understand complicated schemas.

On our evals of complex JSON schema following, our new model `gpt-4o-2024-08-06` with Structured Outputs scores a perfect 100%. In comparison, `gpt-4-0613` scores less than 40%.

■ Prompting Alone ■ Structured Outputs (strict=false) ■ Structured Outputs (strict=true)

Ask ChatGPT



With Structured Outputs, `gpt-4o-2024-08-06` achieves 100% reliability in our evals, perfectly matching the output schemas.

How to use Structured Outputs

We're introducing Structured Outputs in two forms in the API:

1. **Function calling:** Structured Outputs via `tools` is available by setting `strict: true` within your function definition. This feature works with all models that support `tools`, including all models `gpt-4-0613` and `gpt-3.5-turbo-0613` and later. When Structured Outputs are enabled, model outputs will match the supplied tool definition.



JSON



```
1  POST /v1/chat/completions
2  {
3    "model": "gpt-4o-2024-08-06",
4    "messages": [
5      {
6        "role": "system",
7        "content": "You are a helpful assistant. The current date is August 6, 2024."
8      },
9      {
10       "role": "user",
11       "content": "look up all my orders in may of last year that were fulfilled but
12     }
13   ],
14   "tools": [
15     {
16       "type": "function",
17       "function": {
18         "name": "query",
19         "description": "Execute a query.",
20         "strict": true,
21         "parameters": {
22           "type": "object",
```

2. A new option for the `response_format` parameter: developers can now supply a JSON Schema via `json_schema`, a new option for the `response_format` parameter. This is useful when the model is not calling a tool, but rather, responding to the user in a structured way. This feature works with our newest GPT-4o models: `gpt-4o-2024-08-06`, released today, and `gpt-4o-mini-2024-07-18`. When a `response_format` is supplied with `strict: true`, model outputs will match the supplied schema.



Request



```
1 POST /v1/chat/completions
2 {
3   "model": "gpt-4o-2024-08-06",
4   "messages": [
5     {
6       "role": "system",
7       "content": "You are a helpful math tutor."
8     },
9     {
10      "role": "user",
11      "content": "solve  $8x + 31 = 2$ "
12    }
13  ],
14  "response_format": {
15    "type": "json_schema",
16    "json_schema": {
17      "name": "math_response",
18      "strict": true,
19      "schema": {
20        "type": "object",
21        "properties": {
22          "steps": {
23            "type": "array"
```

Safe Structured Outputs

Safety is a top priority for OpenAI—the new Structured Outputs functionality will abide by our existing safety policies and will still allow the model to refuse an unsafe request. To make development simpler, there is a new `refusal` string value on API responses which allows developers to programmatically detect if the model has generated a refusal instead of output matching the schema. When the response does not include a refusal



supplied schema.

JSON



```

1  {
2    "id": "chatcmpl-9nYAG9LPNonX8DAyrkwYfemr3C8HC",
3    "object": "chat.completion",
4    "created": 1721596428,
5    "model": "gpt-4o-2024-08-06",
6    "choices": [
7      {
8        "index": 0,
9        "message": {
10         "role": "assistant",
11         "refusal": "I'm sorry, I cannot assist with that request."
12       },
13       "logprobs": null,
14       "finish_reason": "stop"
15     }
16   ],
17   "usage": {
18     "prompt_tokens": 81,
19     "completion_tokens": 11,
20     "total_tokens": 92
21   },
22   "system_fingerprint": "fp_3407719c7f"
23 }
```

Native SDK support

Our Python and Node SDKs have been updated with native support for Structured Outputs. Supplying a schema for tools or as a response format is as easy as supplying a Pydantic or Zod object, and our SDKs will handle converting the data type to a supported



The following examples show native support for Structured Outputs with function calling.

Python

Node

Python



```
1  from enum import Enum
2  from typing import Union
3
4  from pydantic import BaseModel
5
6  import openai
7  from openai import OpenAI
8
9
10 class Table(str, Enum):
11     orders = "orders"
12     customers = "customers"
13     products = "products"
14
15
16 class Column(str, Enum):
17     id = "id"
18     status = "status"
19     expected_delivery_date = "expected_delivery_date"
20     delivered_at = "delivered_at"
21     shipped_at = "shipped_at"
22     ordered_at = "ordered_at"
```

Native Structured Outputs support is also available for `response_format`.



Python



```
1 from pydantic import BaseModel
2
3 from openai import OpenAI
4
5
6 class Step(BaseModel):
7     explanation: str
8     output: str
9
10
11 class MathResponse(BaseModel):
12     steps: list[Step]
13     final_answer: str
14
15
16 client = OpenAI()
17
18 completion = client.beta.chat.completions.parse(
19     model="gpt-4o-2024-08-06",
20     messages=[
21         {"role": "system", "content": "You are a helpful math tutor."},
22         {"role": "user", "content": "solve 8x + 31 = 2"},
23     ]
24 )
```

Additional use cases

Developers frequently use OpenAI's models to generate structured data for various use cases. Some additional examples include:



user's intent

For example, developers can use Structured Outputs to create code- or UI-generating applications. All of the following examples use the same `response_format`, and can be used to generate varying UIs based on user input.

System

You are a user interface assistant. Your job is to help users visualize their website and app ideas.

Response format

View JSON schema ▾

Assistant

Landing page for a gardener

Sign up screen for an app

Stock price widget

```

1  {
2    "type": "div",
3    "label": "",
4    "children": [
5      {
6        "type": "header",
7        "label": "",
8        "children": [
9          {
10         "type": "div",
11         "label": "Green Thumb Gardening",
12         "children": [],

```



```
15     {
16         "type": "div",
17         "label": "Bringing Life to Your Garden",
18         "children": [],
19         "attributes": [{ "name": "className", "value": "site-tagline" }]
20     }
21 }
```

Welcome to Green Thumb Gardening

Bringing Life to Your Garden

At Green Thumb Gardening, we specialize in transforming your outdoor spaces into beautiful, thriving gardens. Our team has decades of experience in horticulture and landscape design.

Our services

Garden Design

Plant Care & Maintenance

Seasonal Cleanup

Custom Landscaping

Separating a final answer from supporting reasoning or additional commentary



Request

Structured Output

JSON



```
1  {
2    "model": "gpt-4o-2024-08-06",
3    "messages": [
4      {
5        "role": "system",
6        "content": "You are a helpful assistant"
7      },
8      {
9        "role": "user",
10       "content": "9.11 and 9.9 -- which is bigger?"
11     }
12   ],
13   "response_format": {
14     "type": "json_schema",
15     "json_schema": {
16       "name": "reasoning_schema",
17       "strict": true,
18       "schema": {
19         "type": "object",
20         "properties": {
21           "reasoning_steps": {
22             "type": "array",
```

Extracting structured data from unstructured data



Request

Structured Output

JSON



```
1 POST /v1/chat/completions
2 {
3   "model": "gpt-4o-2024-08-06",
4   "messages": [
5     {
6       "role": "system",
7       "content": "Extract action items, due dates, and owners from meeting notes."
8     },
9     {
10      "role": "user",
11      "content": "...meeting notes go here..."
12    }
13  ],
14  "response_format": {
15    "type": "json_schema",
16    "json_schema": {
17      "name": "action_items",
18      "strict": true,
19      "schema": {
20        "type": "object",
21        "properties": {
22          "action_items": {
```

Under the hood



complicated schemas and how best to produce outputs that match them. However, model behavior is inherently non-deterministic—despite this model’s performance improvements (93% on our benchmark), it still did not meet the reliability that developers need to build robust applications. So we also took a deterministic, engineering-based approach to constrain the model’s outputs to achieve 100% reliability.

Constrained decoding

Our approach is based on a technique known as constrained sampling or constrained decoding. By default, when models are sampled to produce outputs, they are entirely unconstrained and can select any token from the vocabulary as the next output. This flexibility is what allows models to make mistakes; for example, they are generally free to sample a curly brace token at any time, even when that would not produce valid JSON. In order to force valid outputs, we constrain our models to only tokens that would be valid according to the supplied schema, rather than all available tokens.

It can be challenging to implement this constraining in practice, since the tokens that are valid differ throughout a model’s output. Let’s say we have the following schema:

JSON





The tokens that are valid at the beginning of the output include things like `{`, `{“`, `{\n`, etc. However, once the model has already sampled `{“val`, then `{` is no longer a valid token. Thus we need to implement dynamic constrained decoding, and determine which tokens are valid after each token is generated, rather than upfront at the beginning of the response.

To do this, we convert the supplied JSON Schema into a context-free grammar (CFG). A grammar is a set of rules that defines a language, and a context-free grammar is a grammar that conforms to specific rules. You can think of JSON and JSON Schema as particular languages with rules to define what is valid within the language. Just as it's not valid in English to have a sentence with no verb, it is not valid in JSON to have a trailing comma.

Thus, for each JSON Schema, we compute a grammar that represents that schema, and pre-process its components to make it easily accessible during model sampling. This is why the first request with a new schema incurs a latency penalty—we must preprocess the schema to generate this artifact that we can use efficiently during sampling.

While sampling, after every token, our inference engine will determine which tokens are valid to be produced next based on the previously generated tokens and the rules within the grammar that indicate which tokens are valid next. We then use this list of tokens to mask the next sampling step, which effectively lowers the probability of invalid tokens to 0. Because we have preprocessed the schema, we can use a cached data structure to do this efficiently, with minimal latency overhead.

Alternate approaches

Alternate approaches to this problem often use finite state machines (FSMs) or regexes (generally implemented with FSMs) for constrained decoding. These function similarly in



broader class of languages than FSMs. In practice, this doesn't matter for very simple schemas like the `value` schema shown above. However, we find that the difference is meaningful for more complex schemas that involve nested or recursive data structures. As an example, FSMs cannot generally express recursive types, which means FSM based approaches may struggle to match parentheses in deeply nested JSON. The following is a sample recursive schema that is supported on the OpenAI API with Structured Outputs but would not be possible to express with a FSM.

JSON



```
1  {
2    "name": "ui",
3    "description": "Dynamically generated UI",
4    "strict": true,
5    "schema": {
6      "type": "object",
7      "properties": {
8        "type": {
9          "type": "string",
10         "description": "The type of the UI component",
11         "enum": ["div", "button", "header", "section", "field", "form"]
12       },
13       "label": {
14         "type": "string",
15         "description": "The label of the UI component, used for buttons or form fields"
16       },
17       "children": {
18         "type": "array",
19         "description": "Nested UI components",
20         "items": {
21           "$ref": "#"
22         }
23       }
24     }
25   }
```



Limitations and restrictions

There are a few limitations to keep in mind when using Structured Outputs:

- Structured Outputs allows only a subset of JSON Schema, detailed [in our docs](#). This helps us ensure the best possible performance.
- The first API response with a new schema will incur additional latency, but subsequent responses will be fast with no latency penalty. This is because during the first request, we process the schema as indicated above and then cache these artifacts for fast reuse later on. Typical schemas take under 10 seconds to process on the first request, but more complex schemas may take up to a minute.
- The model can fail to follow the schema if the model chooses to refuse an unsafe request. If it chooses to refuse, the return message will have the `refusal` boolean set to true to indicate this.
- The model can fail to follow the schema if the generation reaches `max_tokens` or another stop condition before finishing.
- Structured Outputs doesn't prevent all kinds of model mistakes. For example, the model may still make mistakes within the values of the JSON object (e.g., getting a step wrong in a mathematical equation). If developers find mistakes, we recommend providing examples in the system instructions or splitting tasks into simpler subtasks.
- Structured Outputs is not compatible with parallel function calls. When a parallel function call is generated, it may not match supplied schemas. Set `parallel_tool_calls: false` to disable parallel function calling.
- JSON Schemas supplied with Structured Outputs aren't [Zero Data Retention \(ZDR\)](#) eligible.



Availability

Structured Outputs is generally available today in the API.

Structured Outputs with function calling is available on all models that support function calling in the API. This includes our newest models (`gpt-4o` , `gpt-4o-mini`), all models after and including `gpt-4-0613` and `gpt-3.5-turbo-0613` , and any fine-tuned models that support function calling. This functionality is available on the Chat Completions API, Assistants API, and Batch API. Structured Outputs with function calling is also compatible with vision inputs.

Structured Outputs with response formats is available on `gpt-4o-mini` and `gpt-4o-2024-08-06` and any fine tunes based on these models. This functionality is available on the Chat Completions API, Assistants API, and Batch API. Structured Outputs with response formats is also compatible with vision inputs.

By switching to the new `gpt-4o-2024-08-06` , developers save 50% on inputs (\$2.50/1M input tokens) and 33% on outputs (\$10.00/1M output tokens) compared to `gpt-4o-2024-05-13` .

To start using Structured Outputs, check out our [docs](#).

Acknowledgements

Structured Outputs takes inspiration from excellent work from the open source community: namely, the [outlines](#), [jsonformer](#), [instructor](#), [guidance](#), and [lark](#) libraries.



API Platform

2024

Author

Michelle Pokrass

Core contributors

Chris Colby, Melody Guan, Michelle Pokrass, Ted Sanders, Brian Zhang

Acknowledgments

John Allard, Filipe de Avila Belbute Peres, Ilan Bigio, Owen Campbell-Moore, Chen Ding, Atty Eleti, Elie Georges, Katia Gil Guzman, Jeff Harris, Johannes Heidecke, Beth Hoover, Romain Huet, Tomer Kaftan, Jillian Khoo, Karolis Kosas, Ryan Liu, Kevin Lu, Lindsay McCallum, Rohan Nuttall, Joe Palermo, Leher Pathak, Ishaan Singal, Felipe Petroski Such, Freddie Sulit, David Weedon

Our Research	ChatGPT	For Business	Terms & Policies
Research Index	Explore ChatGPT	Business Overview	Terms of Use
Research Overview	Team	Solutions	Privacy Policy
Research Residency	Enterprise	Contact Sales	Security
	Education		Other Policies



OpenAI o3

Download

OpenAI o4-mini

Our Charter

GPT-4o

Sora

Careers

GPT-4o mini

Sora Overview

Brand

Sora

Features

Pricing

Support

Safety

Sora log in ↗

Help Center ↗

Safety Approach

More

Security & Privacy

API Platform

News

Trust & Transparency

Platform Overview

Stories

Pricing

Livestreams

API log in ↗

Podcast

Documentation ↗

Developer Forum ↗



OpenAI © 2015–2025

[Manage Cookies](#)

English United States