

Structured Outputs

=====

Ensure responses adhere to a JSON schema.

Try it out

Try it out in the [Playground](/playground) or generate a ready-to-use schema definition to experiment with structured outputs.

Generate

Introduction

JSON is one of the most widely used formats in the world for applications to exchange data.

Structured Outputs is a feature that ensures the model will always generate responses that adhere to your supplied [JSON Schema](<https://json-schema.org/overview/what-is-jsonschema>), so you don't need to worry about the model omitting a required key, or hallucinating an invalid enum value.

Some benefits of Structured Outputs include:

1. **Reliable type-safety:** No need to validate or retry incorrectly formatted responses
2. **Explicit refusals:** Safety-based model refusals are now programmatically detectable
3. **Simpler prompting:** No need for strongly worded prompts to achieve consistent formatting

In addition to supporting JSON Schema in the REST API, the OpenAI SDKs for [Python](<https://github.com/openai/openai-python/blob/main/helpers.md#structured-outputs-parsing-helpers>) and [JavaScript](<https://github.com/openai/openai-node/blob/master/helpers.md#structured-outputs-parsing-helpers>) also make it easy to define object schemas using [Pydantic](<https://docs.pydantic.dev/latest/>) and [Zod](<https://zod.dev/>) respectively. Below, you can see how to extract information from unstructured text that conforms to a schema defined in code.

Getting a structured response

```
```javascript
import OpenAI from "openai";
import { zodTextFormat } from "openai/helpers/zod";
import { z } from "zod";

const openai = new OpenAI();

const CalendarEvent = z.object({
 name: z.string(),
 date: z.string(),
 participants: z.array(z.string()),
});

const response = await openai.responses.parse({
 model: "gpt-4o-2024-08-06",
```

```

input: [
 { role: "system", content: "Extract the event information." },
 {
 role: "user",
 content: "Alice and Bob are going to a science fair on Friday.",
 },
],
text: {
 format: zodTextFormat(CalendarEvent, "event"),
},
});

const event = response.output_parsed;
"""

"""python
from openai import OpenAI
from pydantic import BaseModel

client = OpenAI()

class CalendarEvent(BaseModel):
 name: str
 date: str
 participants: list[str]

response = client.responses.parse(
 model="gpt-4o-2024-08-06",
 input=[
 {"role": "system", "content": "Extract the event information."},
 {
 "role": "user",
 "content": "Alice and Bob are going to a science fair on Friday.",
 },
],
 text_format=CalendarEvent,
)

event = response.output_parsed
"""

```

### ### Supported models

Structured Outputs is available in our [latest large language models](/docs/models), starting with GPT-4o. Older models like `gpt-4-turbo` and earlier may use [JSON mode](#json-mode) instead.

When to use Structured Outputs via function calling vs via text.format

---

Structured Outputs is available in two forms in the OpenAI API:

1. When using [function calling](/docs/guides/function-calling)
2. When using a `json\_schema` response format

Function calling is useful when you are building an application that bridges the models and functionality of your application.

For example, you can give the model access to functions that query a database in order to build an AI assistant that can help users with their orders, or functions that can interact with the UI.

Conversely, Structured Outputs via `response_format` are more suitable when you want to indicate a structured schema for use when the model responds to the user, rather than when the model calls a tool.

For example, if you are building a math tutoring application, you might want the assistant to respond to your user using a specific JSON Schema so that you can generate a UI that displays different parts of the model's output in distinct ways.

Put simply:

- \* If you are connecting the model to tools, functions, data, etc. in your system, then you should use function calling
- \* If you want to structure the model's output when it responds to the user, then you should use a structured `text.format`

The remainder of this guide will focus on non-function calling use cases in the Responses API. To learn more about how to use Structured Outputs with function calling, check out the [\[Function Calling\]\(/docs/guides/function-calling#function-calling-with-structured-outputs\)](#) guide.

### ### Structured Outputs vs JSON mode

Structured Outputs is the evolution of `[JSON mode](#json-mode)`. While both ensure valid JSON is produced, only Structured Outputs ensure schema adherence. Both Structured Outputs and JSON mode are supported in the Responses API, Chat Completions API, Assistants API, Fine-tuning API and Batch API.

We recommend always using Structured Outputs instead of JSON mode when possible.

However, Structured Outputs with `response_format: {type: "json_schema", ...}` is only supported with the `gpt-4o-mini`, `gpt-4o-mini-2024-07-18`, and `gpt-4o-2024-08-06` model snapshots and later.

Structured Outputs	JSON Mode
--- --- ---	
Outputs valid JSON	Yes Yes
Adheres to schema	Yes (see supported schemas) No
Compatible models	gpt-4o-mini, gpt-4o-2024-08-06, and later gpt-3.5-turbo, gpt-4-* and gpt-4o-* models
Enabling	<code>text: { format: { type: "json_schema", "strict": true, "schema": ... } }</code>   <code>text: { format: { type: "json_object" } }</code>

### Examples

-----

Chain of thought

### ### Chain of thought

You can ask the model to output an answer in a structured, step-by-step way, to guide the user through the solution.

#### Structured Outputs for chain-of-thought math tutoring

```
```javascript
import OpenAI from "openai";
import { zodTextFormat } from "openai/helpers/zod";
import { z } from "zod";

const openai = new OpenAI();

const Step = z.object({
  explanation: z.string(),
  output: z.string(),
});

const MathReasoning = z.object({
  steps: z.array(Step),
  final_answer: z.string(),
});

const response = await openai.responses.parse({
  model: "gpt-4o-2024-08-06",
  input: [
    {
      role: "system",
      content:
        "You are a helpful math tutor. Guide the user through the solution step by step.",
    },
    { role: "user", content: "how can I solve  $8x + 7 = -23$ " },
  ],
  text: {
    format: zodTextFormat(MathReasoning, "math_reasoning"),
  },
});

const math_reasoning = response.output_parsed;
```

```python
from openai import OpenAI
from pydantic import BaseModel

client = OpenAI()

class Step(BaseModel):
    explanation: str
    output: str

class MathReasoning(BaseModel):
```

```
steps: list[Step]
final_answer: str
```

```
response = client.responses.parse(
    model="gpt-4o-2024-08-06",
    input=[
        {
            "role": "system",
            "content": "You are a helpful math tutor. Guide the user through the solution step by
step.",
        },
        {"role": "user", "content": "how can I solve  $8x + 7 = -23$ "},
    ],
    text_format=MathReasoning,
)
```

```
math_reasoning = response.output_parsed
"""
```

```
"""bash
curl https://api.openai.com/v1/responses \
-H "Authorization: Bearer $OPENAI_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "model": "gpt-4o-2024-08-06",
  "input": [
    {
      "role": "system",
      "content": "You are a helpful math tutor. Guide the user through the solution step by step."
    },
    {
      "role": "user",
      "content": "how can I solve  $8x + 7 = -23$ "
    }
  ],
  "text": {
    "format": {
      "type": "json_schema",
      "name": "math_reasoning",
      "schema": {
        "type": "object",
        "properties": {
          "steps": {
            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "explanation": { "type": "string" },
                "output": { "type": "string" }
              },
              "required": ["explanation", "output"],
              "additionalProperties": false
            }
          }
        }
      }
    }
  },
}
```

```

        "final_answer": { "type": "string" }
      },
      "required": ["steps", "final_answer"],
      "additionalProperties": false
    },
    "strict": true
  }
}
'''

```

Example response

```

'''json
{
  "steps": [
    {
      "explanation": "Start with the equation  $8x + 7 = -23$ .",
      "output": " $8x + 7 = -23$ "
    },
    {
      "explanation": "Subtract 7 from both sides to isolate the term with the variable.",
      "output": " $8x = -23 - 7$ "
    },
    {
      "explanation": "Simplify the right side of the equation.",
      "output": " $8x = -30$ "
    },
    {
      "explanation": "Divide both sides by 8 to solve for x.",
      "output": " $x = -30 / 8$ "
    },
    {
      "explanation": "Simplify the fraction.",
      "output": " $x = -15 / 4$ "
    }
  ],
  "final_answer": "x = -15 / 4"
}
'''

```

Structured data extraction

Structured data extraction

You can define structured fields to extract from unstructured input data, such as research papers.

Extracting data from research papers using Structured Outputs

```

'''javascript
import OpenAI from "openai";
import { zodTextFormat } from "openai/helpers/zod";
import { z } from "zod";

```

```

const openai = new OpenAI();

const ResearchPaperExtraction = z.object({
  title: z.string(),
  authors: z.array(z.string()),
  abstract: z.string(),
  keywords: z.array(z.string()),
});

const response = await openai.responses.parse({
  model: "gpt-4o-2024-08-06",
  input: [
    {
      role: "system",
      content:
        "You are an expert at structured data extraction. You will be given unstructured text from a research paper and should convert it into the given structure.",
    },
    { role: "user", content: "..." },
  ],
  text: {
    format: zodTextFormat(ResearchPaperExtraction, "research_paper_extraction"),
  },
});

const research_paper = response.output_parsed;

```

```

"""python
from openai import OpenAI
from pydantic import BaseModel

client = OpenAI()

class ResearchPaperExtraction(BaseModel):
    title: str
    authors: list[str]
    abstract: str
    keywords: list[str]

response = client.responses.parse(
    model="gpt-4o-2024-08-06",
    input=[
        {
            "role": "system",
            "content": "You are an expert at structured data extraction. You will be given unstructured text from a research paper and should convert it into the given structure.",
        },
        {"role": "user", "content": "..."},
    ],
    text_format=ResearchPaperExtraction,
)

```

```
research_paper = response.output_parsed
'''
```

```
```bash
curl https://api.openai.com/v1/responses \
-H "Authorization: Bearer $OPENAI_API_KEY" \
-H "Content-Type: application/json" \
-d '{
 "model": "gpt-4o-2024-08-06",
 "input": [
 {
 "role": "system",
 "content": "You are an expert at structured data extraction. You will be given unstructured
text from a research paper and should convert it into the given structure."
 },
 {
 "role": "user",
 "content": "..."
 }
],
 "text": {
 "format": {
 "type": "json_schema",
 "name": "research_paper_extraction",
 "schema": {
 "type": "object",
 "properties": {
 "title": { "type": "string" },
 "authors": {
 "type": "array",
 "items": { "type": "string" }
 },
 "abstract": { "type": "string" },
 "keywords": {
 "type": "array",
 "items": { "type": "string" }
 }
 },
 "required": ["title", "authors", "abstract", "keywords"],
 "additionalProperties": false
 },
 "strict": true
 }
 }
}'
'''
```

#### Example response

```
```json
{
  "title": "Application of Quantum Algorithms in Interstellar Navigation: A New Frontier",
  "authors": [
    "Dr. Stella Voyager",

```



```

    "Dr. Nova Star",
    "Dr. Lyra Hunter"
  ],
  "abstract": "This paper investigates the utilization of quantum algorithms to improve
interstellar navigation systems. By leveraging quantum superposition and entanglement, our
proposed navigation system can calculate optimal travel paths through space-time anomalies
more efficiently than classical methods. Experimental simulations suggest a significant
reduction in travel time and fuel consumption for interstellar missions.",
  "keywords": [
    "Quantum algorithms",
    "interstellar navigation",
    "space-time anomalies",
    "quantum superposition",
    "quantum entanglement",
    "space travel"
  ]
}

```

UI generation

UI Generation

You can generate valid HTML by representing it as recursive data structures with constraints, like enums.

Generating HTML using Structured Outputs

```

```javascript
import OpenAI from "openai";
import { zodTextFormat } from "openai/helpers/zod";
import { z } from "zod";

const openai = new OpenAI();

const UI = z.lazy(() =>
 z.object({
 type: z.enum(["div", "button", "header", "section", "field", "form"]),
 label: z.string(),
 children: z.array(UI),
 attributes: z.array(
 z.object({
 name: z.string(),
 value: z.string(),
 })
),
 })
);

const response = await openai.responses.parse({
 model: "gpt-4o-2024-08-06",
 input: [
 {
 role: "system",

```

```

 content: "You are a UI generator AI. Convert the user input into a UI.",
 },
 {
 role: "user",
 content: "Make a User Profile Form",
 },
],
text: {
 format: zodTextFormat(UI, "ui"),
},
});

const ui = response.output_parsed;
```

```python
from enum import Enum
from typing import List

from openai import OpenAI
from pydantic import BaseModel

client = OpenAI()

class UIType(str, Enum):
 div = "div"
 button = "button"
 header = "header"
 section = "section"
 field = "field"
 form = "form"

class Attribute(BaseModel):
 name: str
 value: str

class UI(BaseModel):
 type: UIType
 label: str
 children: List["UI"]
 attributes: List[Attribute]

UI.model_rebuild() # This is required to enable recursive types

class Response(BaseModel):
 ui: UI

response = client.responses.parse(
 model="gpt-4o-2024-08-06",
 input=[
 {
 "role": "system",
 "content": "You are a UI generator AI. Convert the user input into a UI.",
 },

```

```

 {"role": "user", "content": "Make a User Profile Form"},
],
 text_format=Response,
)

```

```

ui = response.output_parsed

```

```

```bash
curl https://api.openai.com/v1/responses \
-H "Authorization: Bearer $OPENAI_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "model": "gpt-4o-2024-08-06",
  "input": [
    {
      "role": "system",
      "content": "You are a UI generator AI. Convert the user input into a UI."
    },
    {
      "role": "user",
      "content": "Make a User Profile Form"
    }
  ],
  "text": {
    "format": {
      "type": "json_schema",
      "name": "ui",
      "description": "Dynamically generated UI",
      "schema": {
        "type": "object",
        "properties": {
          "type": {
            "type": "string",
            "description": "The type of the UI component",
            "enum": ["div", "button", "header", "section", "field", "form"]
          },
          "label": {
            "type": "string",
            "description": "The label of the UI component, used for buttons or form fields"
          },
          "children": {
            "type": "array",
            "description": "Nested UI components",
            "items": {"$ref": "#"}
          },
          "attributes": {
            "type": "array",
            "description": "Arbitrary attributes for the UI component, suitable for any element",
            "items": {
              "type": "object",
              "properties": {
                "name": {
                  "type": "string",

```

```

        "description": "The name of the attribute, for example onClick or className"
      },
      "value": {
        "type": "string",
        "description": "The value of the attribute"
      }
    },
    "required": ["name", "value"],
    "additionalProperties": false
  }
},
"required": ["type", "label", "children", "attributes"],
"additionalProperties": false
},
"strict": true
}
}
}

```

Example response

```

```json
{
 "type": "form",
 "label": "User Profile Form",
 "children": [
 {
 "type": "div",
 "label": "",
 "children": [
 {
 "type": "field",
 "label": "First Name",
 "children": [],
 "attributes": [
 {
 "name": "type",
 "value": "text"
 },
 {
 "name": "name",
 "value": "firstName"
 },
 {
 "name": "placeholder",
 "value": "Enter your first name"
 }
]
 }
]
 },
 {
 "type": "field",
 "label": "Last Name",

```

```

 "children": [],
 "attributes": [
 {
 "name": "type",
 "value": "text"
 },
 {
 "name": "name",
 "value": "lastName"
 },
 {
 "name": "placeholder",
 "value": "Enter your last name"
 }
]
 },
 "attributes": []
},
{
 "type": "button",
 "label": "Submit",
 "children": [],
 "attributes": [
 {
 "name": "type",
 "value": "submit"
 }
]
}
],
"attributes": [
 {
 "name": "method",
 "value": "post"
 },
 {
 "name": "action",
 "value": "/submit-profile"
 }
]
}
}

```

## Moderation

### ### Moderation

You can classify inputs on multiple categories, which is a common way of doing moderation.

### Moderation using Structured Outputs

```

```javascript
import OpenAI from "openai";

```

```

import { zodTextFormat } from "openai/helpers/zod";
import { z } from "zod";

const openai = new OpenAI();

const ContentCompliance = z.object({
  is_violating: z.boolean(),
  category: z.enum(["violence", "sexual", "self_harm"]).nullable(),
  explanation_if_violating: z.string().nullable(),
});

const response = await openai.responses.parse({
  model: "gpt-4o-2024-08-06",
  input: [
    {
      "role": "system",
      "content": "Determine if the user input violates specific guidelines and explain if they do."
    },
    {
      "role": "user",
      "content": "How do I prepare for a job interview?"
    }
  ],
  text: {
    format: zodTextFormat(ContentCompliance, "content_compliance"),
  },
});

const compliance = response.output_parsed;

```

```

python
from enum import Enum
from typing import Optional

from openai import OpenAI
from pydantic import BaseModel

client = OpenAI()

class Category(str, Enum):
    violence = "violence"
    sexual = "sexual"
    self_harm = "self_harm"

class ContentCompliance(BaseModel):
    is_violating: bool
    category: Optional[Category]
    explanation_if_violating: Optional[str]

response = client.responses.parse(
    model="gpt-4o-2024-08-06",
    input=[
        {

```

```

        "role": "system",
        "content": "Determine if the user input violates specific guidelines and explain if they
do.",
    },
    { "role": "user", "content": "How do I prepare for a job interview?" },
],
text_format=ContentCompliance,
)

```

```

compliance = response.output_parsed
'''

```

```

'''bash
curl https://api.openai.com/v1/responses \
-H "Authorization: Bearer $OPENAI_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "model": "gpt-4o-2024-08-06",
  "input": [
    {
      "role": "system",
      "content": "Determine if the user input violates specific guidelines and explain if they do."
    },
    {
      "role": "user",
      "content": "How do I prepare for a job interview?"
    }
  ],
  "text": {
    "format": {
      "type": "json_schema",
      "name": "content_compliance",
      "description": "Determines if content is violating specific moderation rules",
      "schema": {
        "type": "object",
        "properties": {
          "is_violating": {
            "type": "boolean",
            "description": "Indicates if the content is violating guidelines"
          },
          "category": {
            "type": ["string", "null"],
            "description": "Type of violation, if the content is violating guidelines. Null otherwise.",
            "enum": ["violence", "sexual", "self_harm"]
          },
          "explanation_if_violating": {
            "type": ["string", "null"],
            "description": "Explanation of why the content is violating"
          }
        },
        "required": ["is_violating", "category", "explanation_if_violating"],
        "additionalProperties": false
      },
      "strict": true
    }
  }
}
'''

```

```
    }  
  }  
}
```

Example response

```
```json  
{
 "is_violating": false,
 "category": null,
 "explanation_if_violating": null
}
```

## How to use Structured Outputs with text.format

---

### Step 1: Define your schema

First you must design the JSON Schema that the model should be constrained to follow. See the [examples](/docs/guides/structured-outputs#examples) at the top of this guide for reference.

While Structured Outputs supports much of JSON Schema, some features are unavailable either for performance or technical reasons. See [here](/docs/guides/structured-outputs#supported-schemas) for more details.

#### #### Tips for your JSON Schema

To maximize the quality of model generations, we recommend the following:

- \* Name keys clearly and intuitively
- \* Create clear titles and descriptions for important keys in your structure
- \* Create and use evals to determine the structure that works best for your use case

### Step 2: Supply your schema in the API call

To use Structured Outputs, simply specify

```
```json  
text: { format: { type: "json_schema", "strict": true, "schema": ... } }
```

For example:

```
```python  
response = client.responses.create(
 model="gpt-4o-2024-08-06",
 input=[
 {"role": "system", "content": "You are a helpful math tutor. Guide the user through the
solution step by step."},
 {"role": "user", "content": "how can I solve $8x + 7 = -23$ "}
],
```



```

text={
 "format": {
 "type": "json_schema",
 "name": "calendar_event",
 "schema": {
 "type": "object",
 "properties": {
 "steps": {
 "type": "array",
 "items": {
 "type": "object",
 "properties": {
 "explanation": {"type": "string"},
 "output": {"type": "string"}
 },
 "required": ["explanation", "output"],
 "additionalProperties": False
 },
 "required": ["steps", "final_answer"],
 "additionalProperties": False
 },
 "final_answer": {"type": "string"}
 },
 "required": ["steps", "final_answer"],
 "additionalProperties": False
 },
 "strict": True
 }
}
)

```

```

print(response.output_text)

```

```

```javascript
const response = await openai.responses.create({
  model: "gpt-4o-2024-08-06",
  input: [
    { role: "system", content: "You are a helpful math tutor. Guide the user through the solution step by step." },
    { role: "user", content: "how can I solve  $8x + 7 = -23$ " }
  ],
  text: {
    format: {
      type: "json_schema",
      name: "math_response",
      schema: {
        type: "object",
        properties: {
          steps: {
            type: "array",
            items: {
              type: "object",
              properties: {
                explanation: { type: "string" },
                output: { type: "string" }
              }
            }
          }
        }
      }
    }
  }
})

```

```

        },
        required: ["explanation", "output"],
        additionalProperties: false
    }
},
final_answer: { type: "string" }
},
required: ["steps", "final_answer"],
additionalProperties: false
},
strict: true
}
}
});

```

```

console.log(response.output_text);

```

```

```bash
curl https://api.openai.com/v1/responses \
-H "Authorization: Bearer $OPENAI_API_KEY" \
-H "Content-Type: application/json" \
-d '{
 "model": "gpt-4o-2024-08-06",
 "input": [
 {
 "role": "system",
 "content": "You are a helpful math tutor. Guide the user through the solution step by step."
 },
 {
 "role": "user",
 "content": "how can I solve $8x + 7 = -23$ "
 }
],
 "text": {
 "format": {
 "type": "json_schema",
 "name": "math_response",
 "schema": {
 "type": "object",
 "properties": {
 "steps": {
 "type": "array",
 "items": {
 "type": "object",
 "properties": {
 "explanation": { "type": "string" },
 "output": { "type": "string" }
 },
 "required": ["explanation", "output"],
 "additionalProperties": false
 }
 },
 "final_answer": { "type": "string" }
 }
 }
 }
 }
}';

```

```

 },
 "required": ["steps", "final_answer"],
 "additionalProperties": false
 },
 "strict": true
}
}
}'
'''

```

**\*\*Note:\*\*** the first request you make with any schema will have additional latency as our API processes the schema, but subsequent requests with the same schema will not have additional latency.

### Step 3: Handle edge cases

In some cases, the model might not generate a valid response that matches the provided JSON schema.

This can happen in the case of a refusal, if the model refuses to answer for safety reasons, or if for example you reach a max tokens limit and the response is incomplete.

```
``javascript
try {
const response = await openai.responses.create({
 model: "gpt-4o-2024-08-06",
 input: [{
 role: "system",
 content: "You are a helpful math tutor. Guide the user through the solution step by step.",
 },
 {
 role: "user",
 content: "how can I solve $8x + 7 = -23$ "
 },
],
max_output_tokens: 50,
text: {
 format: {
 type: "json_schema",
 name: "math_response",
 schema: {
 type: "object",
 properties: {
 steps: {
 type: "array",
 items: {
 type: "object",
 properties: {
 explanation: {
 type: "string"
 },
 output: {
 type: "string"
 }
 }
 }
 }
 }
 }
 }
},
});
} catch (error) {
 console.error("Error:", error);
}
```

```

 },
 required: ["explanation", "output"],
 additionalProperties: false,
 },
 },
 final_answer: {
 type: "string"
 },
 },
 required: ["steps", "final_answer"],
 additionalProperties: false,
},
strict: true,
},
}
});

```

```

if (response.status === "incomplete" && response.incomplete_details.reason ===
"max_output_tokens") {
 // Handle the case where the model did not return a complete response
 throw new Error("Incomplete response");
}

```

```

const math_response = response.output[0].content[0];

```

```

if (math_response.type === "refusal") {
 // handle refusal
 console.log(math_response.refusal);
} else if (math_response.type === "output_text") {
 console.log(math_response.text);
} else {
 throw new Error("No response content");
}
} catch (e) {
 // Handle edge cases
 console.error(e);
}
...

```

```

```python
try:
    response = client.responses.create(
        model="gpt-4o-2024-08-06",
        input=[
            {
                "role": "system",
                "content": "You are a helpful math tutor. Guide the user through the solution step by
step.",
            },
            {"role": "user", "content": "how can I solve  $8x + 7 = -23$ "},
        ],
        text={
            "format": {
                "type": "json_schema",

```

```

        "name": "math_response",
        "strict": True,
        "schema": {
            "type": "object",
            "properties": {
                "steps": {
                    "type": "array",
                    "items": {
                        "type": "object",
                        "properties": {
                            "explanation": {"type": "string"},
                            "output": {"type": "string"},
                        },
                        "required": ["explanation", "output"],
                        "additionalProperties": False,
                    },
                },
                "final_answer": {"type": "string"},
            },
            "required": ["steps", "final_answer"],
            "additionalProperties": False,
        },
        "strict": True,
    },
)
except Exception as e:
    # handle errors like finish_reason, refusal, content_filter, etc.
    pass

```

Refusals with Structured Outputs

When using Structured Outputs with user-generated input, OpenAI models may occasionally refuse to fulfill the request for safety reasons. Since a refusal does not necessarily follow the schema you have supplied in `response_format`, the API response will include a new field called `refusal` to indicate that the model refused to fulfill the request.

When the `refusal` property appears in your output object, you might present the refusal in your UI, or include conditional logic in code that consumes the response to handle the case of a refused request.

```

python
class Step(BaseModel):
    explanation: str
    output: str

class MathReasoning(BaseModel):
    steps: list[Step]
    final_answer: str

completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",
    messages=[

```

```

        {"role": "system", "content": "You are a helpful math tutor. Guide the user through the solution step by step."},
        {"role": "user", "content": "how can I solve  $8x + 7 = -23$ "}
    ],
    response_format=MathReasoning,
)

```

```

math_reasoning = completion.choices[0].message

```

```

# If the model refuses to respond, you will get a refusal message
if (math_reasoning.refusal):
    print(math_reasoning.refusal)
else:
    print(math_reasoning.parsed)
...

```

```

```javascript
const Step = z.object({
 explanation: z.string(),
 output: z.string(),
});

const MathReasoning = z.object({
 steps: z.array(Step),
 final_answer: z.string(),
});

const completion = await openai.beta.chat.completions.parse({
 model: "gpt-4o-2024-08-06",
 messages: [
 { role: "system", content: "You are a helpful math tutor. Guide the user through the solution step by step." },
 { role: "user", content: "how can I solve $8x + 7 = -23$ " },
],
 response_format: zodResponseFormat(MathReasoning, "math_reasoning"),
});

const math_reasoning = completion.choices[0].message

```

```

// If the model refuses to respond, you will get a refusal message
if (math_reasoning.refusal) {
 console.log(math_reasoning.refusal);
} else {
 console.log(math_reasoning.parsed);
}
...

```

The API response from a refusal will look something like this:

```

```json
{
  "id": "resp_1234567890",
  "object": "response",
  "created_at": 1721596428,

```

```

"status": "completed",
"error": null,
"incomplete_details": null,
"input": [],
"instructions": null,
"max_output_tokens": null,
"model": "gpt-4o-2024-08-06",
"output": [{
  "id": "msg_1234567890",
  "type": "message",
  "role": "assistant",
  "content": [
    {
      "type": "refusal",
      "refusal": "I'm sorry, I cannot assist with that request."
    }
  ]
}],
"usage": {
  "input_tokens": 81,
  "output_tokens": 11,
  "total_tokens": 92,
  "output_tokens_details": {
    "reasoning_tokens": 0,
  }
},
}

```

Tips and best practices

Handling user-generated input

If your application is using user-generated input, make sure your prompt includes instructions on how to handle situations where the input cannot result in a valid response.

The model will always try to adhere to the provided schema, which can result in hallucinations if the input is completely unrelated to the schema.

You could include language in your prompt to specify that you want to return empty parameters, or a specific sentence, if the model detects that the input is incompatible with the task.

Handling mistakes

Structured Outputs can still contain mistakes. If you see mistakes, try adjusting your instructions, providing examples in the system instructions, or splitting tasks into simpler subtasks. Refer to the [\[prompt engineering guide\]\(/docs/guides/prompt-engineering\)](#) for more guidance on how to tweak your inputs.

Avoid JSON schema divergence

To prevent your JSON Schema and corresponding types in your programming language from diverging, we strongly recommend using the native Pydantic/zod sdk support.

If you prefer to specify the JSON schema directly, you could add CI rules that flag when either the JSON schema or underlying data objects are edited, or add a CI step that auto-generates the JSON Schema from type definitions (or vice-versa).

Streaming

You can use streaming to process model responses or function call arguments as they are being generated, and parse them as structured data.

That way, you don't have to wait for the entire response to complete before handling it. This is particularly useful if you would like to display JSON fields one by one, or handle function call arguments as soon as they are available.

We recommend relying on the SDKs to handle streaming with Structured Outputs.

```
```python
from typing import List

from openai import OpenAI
from pydantic import BaseModel

class EntitiesModel(BaseModel):
 attributes: List[str]
 colors: List[str]
 animals: List[str]

client = OpenAI()

with client.responses.stream(
 model="gpt-4.1",
 input=[
 {"role": "system", "content": "Extract entities from the input text"},
 {
 "role": "user",
 "content": "The quick brown fox jumps over the lazy dog with piercing blue eyes",
 },
],
 text_format=EntitiesModel,
) as stream:
 for event in stream:
 if event.type == "response.refusal.delta":
 print(event.delta, end="")
 elif event.type == "response.output_text.delta":
 print(event.delta, end="")
 elif event.type == "response.error":
 print(event.error, end="")
 elif event.type == "response.completed":
 print("Completed")
 # print(event.response.output)

 final_response = stream.get_final_response()
 print(final_response)
```



```
```
```

```
```javascript
import { OpenAI } from "openai";
import { zodTextFormat } from "openai/helpers/zod";
import { z } from "zod";

const EntitiesSchema = z.object({
 attributes: z.array(z.string()),
 colors: z.array(z.string()),
 animals: z.array(z.string()),
});

const openai = new OpenAI();
const stream = openai.responses
 .stream({
 model: "gpt-4.1",
 input: [
 { role: "user", content: "What's the weather like in Paris today?" },
],
 text: {
 format: zodTextFormat(EntitiesSchema, "entities"),
 },
 })
 .on("response.refusal.delta", (event) => {
 process.stdout.write(event.delta);
 })
 .on("response.output_text.delta", (event) => {
 process.stdout.write(event.delta);
 })
 .on("response.output_text.done", () => {
 process.stdout.write("\n");
 })
 .on("response.error", (event) => {
 console.error(event.error);
 });

const result = await stream.finalResponse();

console.log(result);
```
```

Supported schemas

Structured Outputs supports a subset of the [JSON Schema](<https://json-schema.org/docs>) language.

Supported types

The following types are supported for Structured Outputs:

- * String
- * Number

- * Boolean
- * Integer
- * Object
- * Array
- * Enum
- * anyOf

Supported properties

In addition to specifying the type of a property, you can specify a selection of additional constraints:

Supported `string` properties:

- * `pattern` — A regular expression that the string must match.
- * `format` — Predefined formats for strings. Currently supported:
 - * `date-time`
 - * `time`
 - * `date`
 - * `duration`
 - * `email`
 - * `hostname`
 - * `ipv4`
 - * `ipv6`
 - * `uuid`

Supported `number` properties:

- * `multipleOf` — The number must be a multiple of this value.
- * `maximum` — The number must be less than or equal to this value.
- * `exclusiveMaximum` — The number must be less than this value.
- * `minimum` — The number must be greater than or equal to this value.
- * `exclusiveMinimum` — The number must be greater than this value.

Supported `array` properties:

- * `minItems` — The array must have at least this many items.
- * `maxItems` — The array must have at most this many items.

Here are some examples on how you can use these type restrictions:

String Restrictions

```
```json
{
 "name": "user_data",
 "strict": true,
 "schema": {
 "type": "object",
 "properties": {
 "name": {
 "type": "string",
 "description": "The name of the user"
 },
 },
 },
}
```

```

 "username": {
 "type": "string",
 "description": "The username of the user. Must start with @",
 "pattern": "^@[a-zA-Z0-9_]+$"
 },
 "email": {
 "type": "string",
 "description": "The email of the user",
 "format": "email"
 }
 },
 "additionalProperties": false,
 "required": [
 "name", "username", "email"
]
}
}

```

## Number Restrictions

```

```json
{
  "name": "weather_data",
  "strict": true,
  "schema": {
    "type": "object",
    "properties": {
      "location": {
        "type": "string",
        "description": "The location to get the weather for"
      },
      "unit": {
        "type": ["string", "null"],
        "description": "The unit to return the temperature in",
        "enum": ["F", "C"]
      },
      "value": {
        "type": "number",
        "description": "The actual temperature value in the location",
        "minimum": -130,
        "maximum": 130
      }
    },
    "additionalProperties": false,
    "required": [
      "location", "unit", "value"
    ]
  }
}

```

Note these constraints are [not yet supported for fine-tuned models](#some-type-specific-keywords-are-not-yet-supported).

Root objects must not be `anyOf` and must be an object

Note that the root level object of a schema must be an object, and not use `anyOf`. A pattern that appears in Zod (as one example) is using a discriminated union, which produces an `anyOf` at the top level. So code such as the following won't work:

```
```javascript
import { z } from 'zod';
import { zodResponseFormat } from 'openai/helpers/zod';

const BaseResponseSchema = z.object({ /* ... */ });
const UnsuccessfulResponseSchema = z.object({ /* ... */ });

const finalSchema = z.discriminatedUnion('status', [
 BaseResponseSchema,
 UnsuccessfulResponseSchema,
]);

// Invalid JSON Schema for Structured Outputs
const json = zodResponseFormat(finalSchema, 'final_schema');
```
```

All fields must be `required`

To use Structured Outputs, all fields or function parameters must be specified as `required`.

```
```json
{
 "name": "get_weather",
 "description": "Fetches the weather in the given location",
 "strict": true,
 "parameters": {
 "type": "object",
 "properties": {
 "location": {
 "type": "string",
 "description": "The location to get the weather for"
 },
 "unit": {
 "type": "string",
 "description": "The unit to return the temperature in",
 "enum": ["F", "C"]
 }
 },
 "additionalProperties": false,
 "required": ["location", "unit"]
 }
}
```
```

Although all fields must be required (and the model will return a value for each parameter), it is possible to emulate an optional parameter by using a union type with `null`.

```

```json
{
 "name": "get_weather",
 "description": "Fetches the weather in the given location",
 "strict": true,
 "parameters": {
 "type": "object",
 "properties": {
 "location": {
 "type": "string",
 "description": "The location to get the weather for"
 },
 "unit": {
 "type": ["string", "null"],
 "description": "The unit to return the temperature in",
 "enum": ["F", "C"]
 }
 },
 "additionalProperties": false,
 "required": [
 "location", "unit"
]
 }
}
```

```

Objects have limitations on nesting depth and size

A schema may have up to 100 object properties total, with up to 5 levels of nesting.

Limitations on total string size

In a schema, total string length of all property names, definition names, enum values, and const values cannot exceed 15,000 characters.

Limitations on enum size

A schema may have up to 500 enum values across all enum properties.

For a single enum property with string values, the total string length of all enum values cannot exceed 7,500 characters when there are more than 250 enum values.

`additionalProperties: false` must always be set in objects

`additionalProperties` controls whether it is allowable for an object to contain additional keys / values that were not defined in the JSON Schema.

Structured Outputs only supports generating specified keys / values, so we require developers to set `additionalProperties: false` to opt into Structured Outputs.

```

```json
{
 "name": "get_weather",
 "description": "Fetches the weather in the given location",

```

```

"strict": true,
"schema": {
 "type": "object",
 "properties": {
 "location": {
 "type": "string",
 "description": "The location to get the weather for"
 },
 "unit": {
 "type": "string",
 "description": "The unit to return the temperature in",
 "enum": ["F", "C"]
 }
 },
 "additionalProperties": false,
 "required": [
 "location", "unit"
]
}
}

```

#### #### Key ordering

When using Structured Outputs, outputs will be produced in the same order as the ordering of keys in the schema.

#### #### Some type-specific keywords are not yet supported

Specifically:

- \* \*\*For objects:\*\* `unevaluatedProperties`, `propertyNames`, `minProperties`, `maxProperties`
- \* \*\*For arrays:\*\* `unevaluatedItems`, `contains`, `minContains`, `maxContains`, `uniqueItems`
- \* \*\*Composition:\*\* `allOf`, `not`, `dependentRequired`, `dependentSchemas`, `if`, `then`, `else`

For fine-tuned models, we additionally do not support the following:

- \* \*\*For strings:\*\* `minLength`, `maxLength`, `pattern`, `format`
- \* \*\*For numbers:\*\* `minimum`, `maximum`, `multipleOf`
- \* \*\*For objects:\*\* `patternProperties`
- \* \*\*For arrays:\*\* `minItems`, `maxItems`

If you turn on Structured Outputs by supplying `strict: true` and call the API with an unsupported JSON Schema, you will receive an error.

#### #### For `anyOf`, the nested schemas must each be a valid JSON Schema per this subset

Here's an example supported anyOf schema:

```

```json
{
  "type": "object",
  "properties": {
    "item": {

```

```

"anyOf": [
  {
    "type": "object",
    "description": "The user object to insert into the database",
    "properties": {
      "name": {
        "type": "string",
        "description": "The name of the user"
      },
      "age": {
        "type": "number",
        "description": "The age of the user"
      }
    },
    "additionalProperties": false,
    "required": [
      "name",
      "age"
    ]
  },
  {
    "type": "object",
    "description": "The address object to insert into the database",
    "properties": {
      "number": {
        "type": "string",
        "description": "The number of the address. Eg. for 123 main st, this would be
123"
      },
      "street": {
        "type": "string",
        "description": "The street name. Eg. for 123 main st, this would be main st"
      },
      "city": {
        "type": "string",
        "description": "The city of the address"
      }
    },
    "additionalProperties": false,
    "required": [
      "number",
      "street",
      "city"
    ]
  }
]
},
"additionalProperties": false,
"required": [
  "item"
]
}

```

Definitions are supported

You can use definitions to define subschemas which are referenced throughout your schema. The following is a simple example.

```
```json
{
 "type": "object",
 "properties": {
 "steps": {
 "type": "array",
 "items": {
 "$ref": "#/$defs/step"
 }
 },
 "final_answer": {
 "type": "string"
 }
 },
 "$defs": {
 "step": {
 "type": "object",
 "properties": {
 "explanation": {
 "type": "string"
 },
 "output": {
 "type": "string"
 }
 },
 "required": [
 "explanation",
 "output"
],
 "additionalProperties": false
 }
 },
 "required": [
 "steps",
 "final_answer"
],
 "additionalProperties": false
}
```
```

Recursive schemas are supported

Sample recursive schema using `#` to indicate root recursion.

```
```json
{
 "name": "ui",
 "description": "Dynamically generated UI",

```



```

"strict": true,
"schema": {
 "type": "object",
 "properties": {
 "type": {
 "type": "string",
 "description": "The type of the UI component",
 "enum": ["div", "button", "header", "section", "field", "form"]
 },
 "label": {
 "type": "string",
 "description": "The label of the UI component, used for buttons or form fields"
 },
 "children": {
 "type": "array",
 "description": "Nested UI components",
 "items": {
 "$ref": "#"
 }
 },
 "attributes": {
 "type": "array",
 "description": "Arbitrary attributes for the UI component, suitable for any element",
 "items": {
 "type": "object",
 "properties": {
 "name": {
 "type": "string",
 "description": "The name of the attribute, for example onClick or className"
 },
 "value": {
 "type": "string",
 "description": "The value of the attribute"
 }
 },
 "additionalProperties": false,
 "required": ["name", "value"]
 }
 }
 },
 "required": ["type", "label", "children", "attributes"],
 "additionalProperties": false
}
}

```

Sample recursive schema using explicit recursion:

```

```json
{
  "type": "object",
  "properties": {
    "linked_list": {
      "$ref": "#/$defs/linked_list_node"
    }
  }
}

```

```

    }
  },
  "$defs": {
    "linked_list_node": {
      "type": "object",
      "properties": {
        "value": {
          "type": "number"
        },
        "next": {
          "anyOf": [
            {
              "$ref": "#/$defs/linked_list_node"
            },
            {
              "type": "null"
            }
          ]
        }
      }
    },
    "additionalProperties": false,
    "required": [
      "next",
      "value"
    ]
  }
},
"additionalProperties": false,
"required": [
  "linked_list"
]
}

```

JSON mode

JSON mode is a more basic version of the Structured Outputs feature. While JSON mode ensures that model output is valid JSON, Structured Outputs reliably matches the model's output to the schema you specify. We recommend you use Structured Outputs if it is supported for your use case.

When JSON mode is turned on, the model's output is ensured to be valid JSON, except for in some edge cases that you should detect and handle appropriately.

To turn on JSON mode with the Responses API you can set the `text.format` to `{ "type": "json_object" }`. If you are using function calling, JSON mode is always turned on.

Important notes:

- * When using JSON mode, you must always instruct the model to produce JSON via some message in the conversation, for example via your system message. If you don't include an explicit instruction to generate JSON, the model may generate an unending stream of whitespace and the request may run continually until it reaches the token limit. To help ensure

you don't forget, the API will throw an error if the string "JSON" does not appear somewhere in the context.

- * JSON mode will not guarantee the output matches any specific schema, only that it is valid and parses without errors. You should use Structured Outputs to ensure it matches your schema, or if that is not possible, you should use a validation library and potentially retries to ensure that the output matches your desired schema.

- * Your application must detect and handle the edge cases that can result in the model output not being a complete JSON object (see below)

Handling edge cases

```
```javascript
const we_did_not_specify_stop_tokens = true;

try {
 const response = await openai.responses.create({
 model: "gpt-3.5-turbo-0125",
 input: [
 {
 role: "system",
 content: "You are a helpful assistant designed to output JSON.",
 },
 { role: "user", content: "Who won the world series in 2020? Please respond in the format {winner: ...}" },
],
 text: { format: { type: "json_object" } },
 });

 // Check if the conversation was too long for the context window, resulting in incomplete JSON
 if (response.status === "incomplete" && response.incomplete_details.reason === "max_output_tokens") {
 // your code should handle this error case
 }

 // Check if the OpenAI safety system refused the request and generated a refusal instead
 if (response.output[0].content[0].type === "refusal") {
 // your code should handle this error case
 // In this case, the .content field will contain the explanation (if any) that the model generated for why it is refusing
 console.log(response.output[0].content[0].refusal)
 }

 // Check if the model's output included restricted content, so the generation of JSON was halted and may be partial
 if (response.status === "incomplete" && response.incomplete_details.reason === "content_filter") {
 // your code should handle this error case
 }

 if (response.status === "completed") {
 // In this case the model has either successfully finished generating the JSON object according to your schema, or the model generated one of the tokens you provided as a "stop token"
 }
}
```

```

 if (we_did_not_specify_stop_tokens) {
 // If you didn't specify any stop tokens, then the generation is complete and the content
 key will contain the serialized JSON object
 // This will parse successfully and should now contain {"winner": "Los Angeles Dodgers"}
 console.log(JSON.parse(response.output_text))
 } else {
 // Check if the response.output_text ends with one of your stop tokens and handle
 appropriately
 }
} catch (e) {
 // Your code should handle errors here, for example a network error calling the API
 console.error(e)
}
...

```

```

```python
we_did_not_specify_stop_tokens = True

```

```

try:
    response = client.responses.create(
        model="gpt-3.5-turbo-0125",
        input=[
            {"role": "system", "content": "You are a helpful assistant designed to output JSON."},
            {"role": "user", "content": "Who won the world series in 2020? Please respond in the
format {winner: ...}"}
        ],
        text={"format": {"type": "json_object"}}
    )

```

```

    # Check if the conversation was too long for the context window, resulting in incomplete
    JSON

```

```

    if response.status == "incomplete" and response.incomplete_details.reason ==
    "max_output_tokens":
        # your code should handle this error case
        pass

```

```

    # Check if the OpenAI safety system refused the request and generated a refusal instead
    if response.output[0].content[0].type == "refusal":
        # your code should handle this error case
        # In this case, the .content field will contain the explanation (if any) that the model
        generated for why it is refusing
        print(response.output[0].content[0]["refusal"])

```

```

    # Check if the model's output included restricted content, so the generation of JSON was
    halted and may be partial

```

```

    if response.status == "incomplete" and response.incomplete_details.reason ==
    "content_filter":
        # your code should handle this error case
        pass

```

```

    if response.status == "completed":

```

In this case the model has either successfully finished generating the JSON object according to your schema, or the model generated one of the tokens you provided as a "stop token"

```
if we_did_not_specify_stop_tokens:
    # If you didn't specify any stop tokens, then the generation is complete and the content
    key will contain the serialized JSON object
    # This will parse successfully and should now contain '{"winner": "Los Angeles
    Dodgers"}'
    print(response.output_text)
else:
    # Check if the response.output_text ends with one of your stop tokens and handle
    appropriately
    pass
except Exception as e:
    # Your code should handle errors here, for example a network error calling the API
    print(e)
...
```

Resources

To learn more about Structured Outputs, we recommend browsing the following resources:

- * Check out our [introductory cookbook](https://cookbook.openai.com/examples/structured_outputs_intro) on Structured Outputs
- * Learn [how to build multi-agent systems](https://cookbook.openai.com/examples/structured_outputs_multi_agent) with Structured Outputs

Was this page useful?