Prompt generation

Copy page

Generate prompts and schemas in Playground.

The **Generate** button in the Playground lets you generate prompts, functions, and schemas from just a description of your task. This guide will walk through exactly how it works.

Overview

Creating prompts and schemas from scratch can be time-consuming, so generating them can help you get started quickly. The Generate button uses two main approaches:

1. **Prompts:** We use **meta-prompts** that incorporate best practices to generate or improve prompts.
2. **Schemas:** We use **meta-schemas** that produce valid JSON and function syntax.

While we currently use meta prompts and schemas, we may integrate more advanced techniques in the future like DSPy and "Gradient Descent".

Prompts

A **meta-prompt** instructs the model to create a good prompt based on your task description or improve an existing one. The meta-prompts in the Playground draw from our prompt engineering best practices and real-world experience with users.

We use specific meta-prompts for different output types, like audio, to ensure the generated prompts meet the expected format.

Meta-prompts

Text-out   Audio-out

Text meta-prompt   python

```python
from openai import OpenAI

client = OpenAI()

META_PROMPT = """
Given a task description or existing prompt, produce a detailed system prompt to guide a lar

# Guidelines
```

```

- Understand the Task: Grasp the main objective, goals, requirements, constraints, and expec
- Minimal Changes: If an existing prompt is provided, improve it only if it's simple. For co
- Reasoning Before Conclusions**: Encourage reasoning steps before any conclusions are reach
    - Reasoning Order: Call out reasoning portions of the prompt and conclusion parts (speci
    - Conclusion, classifications, or results should ALWAYS appear last.
- Examples: Include high-quality examples if helpful, using placeholders [in brackets] for c
   - What kinds of examples may need to be included, how many, and whether they are complex
- Clarity and Conciseness: Use clear, specific language. Avoid unnecessary instructions or b
- Formatting: Use markdown features for readability. DO NOT USE ``` CODE BLOCKS UNLESS SPECI
- Preserve User Content: If the input task or prompt includes extensive guidelines or exampl
- Constants: DO include constants in the prompt, as they are not susceptible to prompt injec
- Output Format: Explicitly the most appropriate output format, in detail. This should inclu
    - For tasks outputting well-defined or structured data (classification, JSON, etc.) bias
    - JSON should never be wrapped in code blocks (```) unless explicitly requested.

The final prompt you output should adhere to the following structure below. Do not include a

[Concise instruction describing the task - this should be the first line in the prompt, no s

[Additional details as needed.]

[Optional sections with headings or bullet points for detailed steps.]

# Steps [optional]

[optional: a detailed breakdown of the steps necessary to accomplish the task]

# Output Format

[Specifically call out how the output should be formatted, be it response length, structure

# Examples [optional]

[Optional: 1-3 well-defined examples with placeholders if necessary. Clearly mark where exam
[If the examples are shorter than what a realistic example is expected to be, make a referen

# Notes [optional]

[optional: edge cases, details, and an area to call or repeat out specific important conside
""".strip()

def generate_prompt(task_or_prompt: str):
    completion = client.chat.completions.create(
        model="gpt-4o",
        messages=[
            {
                "role": "system",
                "content": META_PROMPT,
            },
            {
                "role": "user",
                "content": "Task, Goal, or Current Prompt:\n" + task_or_prompt,
            },
        ],
    )
```

```
    return completion.choices[0].message.content
```

Prompt edits

To edit prompts, we use a slightly modified meta-prompt. While direct edits are straightforward to apply, identifying necessary changes for more open-ended revisions can be challenging. To address this, we include a **reasoning section** at the beginning of the response. This section helps guide the model in determining what changes are needed by evaluating the existing prompt's clarity, chain-of-thought ordering, overall structure, and specificity, among other factors. The reasoning section makes suggestions for improvements and is then parsed out from the final response.

Text-out   Audio-out

Text meta-prompt for edits   python

```python
from openai import OpenAI

client = OpenAI()

META_PROMPT = """
Given a current prompt and a change description, produce a detailed system prompt to guide a

Your final output will be the full corrected prompt verbatim. However, before that, at the v
<reasoning>
- Simple Change: (yes/no) Is the change description explicit and simple? (If so, skip the re
- Reasoning: (yes/no) Does the current prompt use reasoning, analysis, or chain of thought?
    - Identify: (max 10 words) if so, which section(s) utilize reasoning?
    - Conclusion: (yes/no) is the chain of thought used to determine a conclusion?
    - Ordering: (before/after) is the chain of though located before or after
- Structure: (yes/no) does the input prompt have a well defined structure
- Examples: (yes/no) does the input prompt have few-shot examples
    - Representative: (1-5) if present, how representative are the examples?
- Complexity: (1-5) how complex is the input prompt?
    - Task: (1-5) how complex is the implied task?
    - Necessity: ()
- Specificity: (1-5) how detailed and specific is the prompt? (not to be confused with lengt
- Prioritization: (list) what 1-3 categories are the MOST important to address.
- Conclusion: (max 30 words) given the previous assessment, give a very concise, imperative
</reasoning>

# Guidelines

- Understand the Task: Grasp the main objective, goals, requirements, constraints, and expec
- Minimal Changes: If an existing prompt is provided, improve it only if it's simple. For co
- Reasoning Before Conclusions**: Encourage reasoning steps before any conclusions are reach
    - Reasoning Order: Call out reasoning portions of the prompt and conclusion parts (speci
    - Conclusion, classifications, or results should ALWAYS appear last.
- Examples: Include high-quality examples if helpful, using placeholders [in brackets] for c
```

```
   - What kinds of examples may need to be included, how many, and whether they are complex
- Clarity and Conciseness: Use clear, specific language. Avoid unnecessary instructions or b
- Formatting: Use markdown features for readability. DO NOT USE ``` CODE BLOCKS UNLESS SPECI
- Preserve User Content: If the input task or prompt includes extensive guidelines or exampl
- Constants: DO include constants in the prompt, as they are not susceptible to prompt injec
- Output Format: Explicitly the most appropriate output format, in detail. This should inclu
    - For tasks outputting well-defined or structured data (classification, JSON, etc.) bias
    - JSON should never be wrapped in code blocks (```) unless explicitly requested.

The final prompt you output should adhere to the following structure below. Do not include a

[Concise instruction describing the task - this should be the first line in the prompt, no s

[Additional details as needed.]

[Optional sections with headings or bullet points for detailed steps.]

# Steps [optional]

[optional: a detailed breakdown of the steps necessary to accomplish the task]

# Output Format

[Specifically call out how the output should be formatted, be it response length, structure

# Examples [optional]

[Optional: 1-3 well-defined examples with placeholders if necessary. Clearly mark where exam
[If the examples are shorter than what a realistic example is expected to be, make a referen

# Notes [optional]

[optional: edge cases, details, and an area to call or repeat out specific important conside
[NOTE: you must start with a <reasoning> section. the immediate next token you produce shoul
""".strip()

def generate_prompt(task_or_prompt: str):
    completion = client.chat.completions.create(
        model="gpt-4o",
        messages=[
            {
                "role": "system",
                "content": META_PROMPT,
            },
            {
                "role": "user",
                "content": "Task, Goal, or Current Prompt:\n" + task_or_prompt,
            },
        ],
    )

    return completion.choices[0].message.content
```

Schemas

Structured Outputs schemas and function schemas are themselves JSON objects, so we leverage Structured Outputs to generate them. This requires defining a schema for the desired output, which in this case is itself a schema. To do this, we use a self-describing schema – a **meta-schema**.

Because the `parameters` field in a function schema is itself a schema, we use the same meta-schema to generate functions.

Defining a constrained meta-schema

Structured Outputs supports two modes: `strict=true` and `strict=false`. Both modes use the same model trained to follow the provided schema, but only "strict mode" guarantees perfect adherence through constrained sampling.

Our goal is to generate schemas for strict mode using strict mode itself. However, the official meta-schemas provided by the JSON Schema Specification rely on features not currently supported in strict mode. This poses challenges that affect both input and output schemas.

1. **Input schema:** We can't use unsupported features in the input schema to describe the output schema.
2. **Output schema:** The generated schema must not include unsupported features.

Because we need to generate new keys in the output schema, the input meta-schema must use `additionalProperties`. This means we can't currently use strict mode to generate schemas. However, we still want the generated schema to conform to strict mode constraints.

To overcome this limitation, we define a **pseudo-meta-schema** — a meta-schema that uses features not supported in strict mode to describe only the features that are supported in strict mode. Essentially, this approach steps outside strict mode for the meta-schema definition while still ensuring that the generated schemas adhere to strict mode constraints.

DEEP DIVE

How we designed the pseudo-meta-schema

Output cleaning

Strict mode guarantees perfect schema adherence. Because we can't use it during generation, however, we need to validate and transform the output after generating it.

After generating a schema, we perform the following steps:

1. **Set** `additionalProperties` **to** `false` for all objects.
2. **Mark all properties as required.**

3. **For structured output schemas,** wrap them in `json_schema` object.
4. **For functions,** wrap them in a `function` object.

ⓘ The Realtime API function object differs slightly from the Chat Completions API, but uses the same schema.

Meta-schemas

Each meta-schema has a corresponding prompt which includes few-shot examples. When combined with the reliability of Structured Outputs — even without strict mode — we were able to use `gpt-4o-mini` for schema generation.

Structured output schema **Function schema**

Structured output meta-schema python

```python
from openai import OpenAI
import json

client = OpenAI()

META_SCHEMA = {
  "name": "metaschema",
  "schema": {
    "type": "object",
    "properties": {
      "name": {
        "type": "string",
        "description": "The name of the schema"
      },
      "type": {
        "type": "string",
        "enum": [
          "object",
          "array",
          "string",
          "number",
          "boolean",
          "null"
        ]
      },
      "properties": {
        "type": "object",
        "additionalProperties": {
          "$ref": "#/$defs/schema_definition"
        }
      },
      "items": {
        "anyOf": [
          {
            "$ref": "#/$defs/schema_definition"
          },
          {
            "type": "array",
            "items": {
              "$ref": "#/$defs/schema_definition"
            }
          }
        ]
      },
      "required": {
        "type": "array",
        "items": {
          "type": "string"
        }
      },
      "additionalProperties": {
        "type": "boolean"
      }
    },
    "required": [
      "type"
    ],
    "additionalProperties": False,
    "if": {
      "properties": {
        "type": {
          "const": "object"
        }
      }
    },
    "then": {
      "required": [
        "properties"
      ]
    },
    "$defs": {
      "schema_definition": {
        "type": "object",
        "properties": {
          "type": {
            "type": "string",
            "enum": [
              "object",
              "array",
              "string",
              "number",
              "boolean",
              "null"
            ]
          },
          "properties": {
            "type": "object",
            "additionalProperties": {
              "$ref": "#/$defs/schema_definition"
            }
          },
            "items": {
              "anyOf": [
                {
                  "$ref": "#/$defs/schema_definition"
                },
                {
                  "type": "array",
                  "items": {
                    "$ref": "#/$defs/schema_definition"
                  }
                }
              ]
            },
            "required": {
              "type": "array",
              "items": {
                "type": "string"
              }
            },
            "additionalProperties": {
              "type": "boolean"
            }
          },
          "required": [
            "type"
          ],
          "additionalProperties": False,
          "if": {
            "properties": {
              "type": {
                "const": "object"
              }
            }
          },
          "then": {
            "required": [
              "properties"
            ]
          }
        }
      }
    }
}

META_PROMPT = """
# Instructions
Return a valid schema for the described JSON.

You must also make sure:
- all fields in an object are set as required
- I REPEAT, ALL FIELDS MUST BE MARKED AS REQUIRED
- all objects must have additionalProperties set to false
    - because of this, some cases like "attributes" or "metadata" properties that would nor
- all objects must have properties defined
- field order matters. any form of "thinking" or "explanation" should come before the concl
- $defs must be defined under the schema param
```

```

Notable keywords NOT supported include:
- For strings: minLength, maxLength, pattern, format
- For numbers: minimum, maximum, multipleOf
- For objects: patternProperties, unevaluatedProperties, propertyNames, minProperties, maxP
- For arrays: unevaluatedItems, contains, minContains, maxContains, minItems, maxItems, uni

Other notes:
- definitions and recursion are supported
- only if necessary to include references e.g. "$defs", it must be inside the "schema" obje

# Examples
Input: Generate a math reasoning schema with steps and a final answer.
Output: {
    "name": "math_reasoning",
    "type": "object",
    "properties": {
        "steps": {
            "type": "array",
            "description": "A sequence of steps involved in solving the math problem.",
            "items": {
                "type": "object",
                "properties": {
                    "explanation": {
                        "type": "string",
                        "description": "Description of the reasoning or method used in this
                    },
                    "output": {
                        "type": "string",
                        "description": "Result or outcome of this specific step."
                    }
                },
                "required": [
                    "explanation",
                    "output"
                ],
                "additionalProperties": false
            }
        },
        "final_answer": {
            "type": "string",
            "description": "The final solution or answer to the math problem."
        }
    },
    "required": [
        "steps",
        "final_answer"
    ],
    "additionalProperties": false
}

Input: Give me a linked list
Output: {
    "name": "linked_list",
    "type": "object",
    "properties": {
        "linked_list": {
            "$ref": "#/$defs/linked_list_node",
            "description": "The head node of the linked list."
        }
    },
    "$defs": {
        "linked_list_node": {
            "type": "object",
            "description": "Defines a node in a singly linked list.",
            "properties": {
                "value": {
                    "type": "number",
                    "description": "The value stored in this node."
                },
                "next": {
                    "anyOf": [
                        {
                            "$ref": "#/$defs/linked_list_node"
                        },
                        {
                            "type": "null"
                        }
                    ],
                    "description": "Reference to the next node; null if it is the last node
                }
            },
            "required": [
                "value",
                "next"
            ],
            "additionalProperties": false
        }
    },
    "required": [
        "linked_list"
    ],
    "additionalProperties": false
}

Input: Dynamically generated UI
Output: {
    "name": "ui",
    "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 ele
                "items": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "description": "The name of the attribute, for example onClick or c
                        },
                        "value": {
                            "type": "string",
                            "description": "The value of the attribute"
                        }
                    },
                    "required": [
                        "name",
                        "value"
                    ],
                    "additionalProperties": false
                }
            }
        },
        "required": [
            "type",
            "label",
            "children",
            "attributes"
        ],
        "additionalProperties": false
    }
    """.strip()

    def generate_schema(description: str):
        completion = client.chat.completions.create(
            model="gpt-4o-mini",
            response_format={"type": "json_schema", "json_schema": META_SCHEMA},
            messages=[
                {
                    "role": "system",
                    "content": META_PROMPT,
                },
                {
                    "role": "user",
                    "content": "Description:\n" + description,
            },
        ],
    )

    return json.loads(completion.choices[0].message.content)
```