Introducing updates to our 2.5 family of thinking models.

Learn more (https://ai.google.dev/gemini-api/docs/models)

Structured output

You can configure Gemini for structured output instead of unstructured text, allowing precise extraction and standardization of information for further processing. For example, you can use structured output to extract information from resumes, standardize them to build a structured database.

Gemini can generate either JSON (/gemini-api/docs/structured-output#generating-json) or enum values (/gemini-api/docs/structured-output#generating-enums) as structured output.

Generating JSON

There are two ways to generate JSON using the Gemini API:

- Configure a schema on the model
- Provide a schema in a text prompt

Configuring a schema on the model is the **recommended** way to generate JSON, because it constrains the model to output JSON.

Configuring a schema (recommended)

To constrain the model to generate JSON, configure a `responseSchema`. The model will then respond to any prompt with JSON-formatted output.

Python (#python) JavaScript (#javascript) Go (#go) REST (#rest)

```
curl "https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-fl
-H 'Content-Type: application/json' \
-d '{
```

```
        "contents": [{
          "parts":[
            { "text": "List a few popular cookie recipes, and include the amc
          ]
        }],
        "generationConfig": {
          "responseMimeType": "application/json",
          "responseSchema": {
            "type": "ARRAY",
            "items": {
              "type": "OBJECT",
              "properties": {
                "recipeName": { "type": "STRING" },
                "ingredients": {
                  "type": "ARRAY",
                  "items": { "type": "STRING" }
                }
              },
              "propertyOrdering": ["recipeName", "ingredients"]
            }
          }
        }
 }' 2> /dev/null | head
```

The output might look like this:

```
[
  {
    "recipeName": "Chocolate Chip Cookies",
    "ingredients": [
      "1 cup (2 sticks) unsalted butter, softened",
      "3/4 cup granulated sugar",
      "3/4 cup packed brown sugar",
      "1 teaspoon vanilla extract",
      "2 large eggs",
      "2 1/4 cups all-purpose flour",
      "1 teaspoon baking soda",
      "1 teaspoon salt",
      "2 cups chocolate chips"
    ]
  },
  ...
```

```
]
```

Providing a schema in a text prompt

Instead of configuring a schema, you can supply a schema as natural language or pseudo-code in a text prompt. This method is **not recommended**, because it might produce lower quality output, and because the model is not constrained to follow the schema.

Warning: Don't provide a schema in a text prompt if you're configuring a **`responseSchema`**. This can produce unexpected or low quality results.

Here's a generic example of a schema provided in a text prompt:

```
List a few popular cookie recipes, and include the amounts of ingredients.

Produce JSON matching this specification:

Recipe = { "recipeName": string, "ingredients": array<string> }
Return: array<Recipe>
```

Since the model gets the schema from text in the prompt, you might have some flexibility in how you represent the schema. But when you supply a schema inline like this, the model is not actually constrained to return JSON. For a more deterministic, higher quality response, configure a schema on the model, and don't duplicate the schema in the text prompt.

Generating enum values

In some cases you might want the model to choose a single option from a list of options. To implement this behavior, you can pass an *enum* in your schema. You can use an enum option anywhere you could use a **`string`** in the **`responseSchema`**, because an enum is an array of strings. Like a JSON schema, an enum lets you constrain model output to meet the requirements of your application.

For example, assume that you're developing an application to classify musical instruments into one of five categories: "**`Percussion`**", "**`String`**", "**`Woodwind`**", "**`Brass`**", or ""**`Keyboard`**"". You

could create an enum to help with this task.

In the following example, you pass an enum as the `responseSchema`, constraining the model to choose the most appropriate option.

Python
(#python)

```
from google import genai
import enum

class Instrument(enum.Enum):
  PERCUSSION = "Percussion"
  STRING = "String"
  WOODWIND = "Woodwind"
  BRASS = "Brass"
  KEYBOARD = "Keyboard"

client = genai.Client(api_key="GEMINI_API_KEY")
response = client.models.generate_content(
    model='gemini-2.5-flash',
    contents='What type of instrument is an oboe?',
    config={
        'response_mime_type': 'text/x.enum',
        'response_schema': Instrument,
    },
)

print(response.text)
# Woodwind
```

The Python library will translate the type declarations for the API. However, the API accepts a subset of the OpenAPI 3.0 schema (Schema (https://ai.google.dev/api/caching#schema)).

There are two other ways to specify an enumeration. You can use a `Literal` (https://docs.pydantic.dev/1.10/usage/types/#literal-type):

Python
(#python)

```
Literal["Percussion", "String", "Woodwind", "Brass", "Keyboard"]
```

And you can also pass the schema as JSON:

Python
(#python)

```
from google import genai

client = genai.Client(api_key="GEMINI_API_KEY")
response = client.models.generate_content(
    model='gemini-2.5-flash',
    contents='What type of instrument is an oboe?',
    config={
        'response_mime_type': 'text/x.enum',
        'response_schema': {
            "type": "STRING",
            "enum": ["Percussion", "String", "Woodwind", "Brass", "Keyboard
        },
    },
)

print(response.text)
# Woodwind
```

Beyond basic multiple choice problems, you can use an enum anywhere in a JSON schema. For example, you could ask the model for a list of recipe titles and use a `Grade` enum to give each title a popularity grade:

Python
(#python)

```
from google import genai

import enum
from pydantic import BaseModel

class Grade(enum.Enum):
    A_PLUS = "a+"
    A = "a"
    B = "b"
```

```
    C = "c"
    D = "d"
    F = "f"

class Recipe(BaseModel):
  recipe_name: str
  rating: Grade

client = genai.Client(api_key="GEMINI_API_KEY")
response = client.models.generate_content(
    model='gemini-2.5-flash',
    contents='List 10 home-baked cookie recipes and give them grades based
    config={
        'response_mime_type': 'application/json',
        'response_schema': list[Recipe],
    },
)

print(response.text)
```

The response might look like this:

```
[
  {
    "recipe_name": "Chocolate Chip Cookies",
    "rating": "a+"
  },
  {
    "recipe_name": "Peanut Butter Cookies",
    "rating": "a"
  },
  {
    "recipe_name": "Oatmeal Raisin Cookies",
    "rating": "b"
  },
  ...
]
```

About JSON schemas

Configuring the model for JSON output using `responseSchema` parameter relies on `Schema` object to define its structure. This object represents a select subset of the OpenAPI 3.0 Schema object (https://spec.openapis.org/oas/v3.0.3#schema-object), and also adds a `propertyOrdering` field.

Tip: On Python, when you use a Pydantic model, you don't need to directly work with `Schema` objects, as it gets automatically converted to the corresponding JSON schema. To learn more, see JSON schemas in Python (#schemas-in-python).

Here's a pseudo-JSON representation of all the `Schema` fields:

```
{
  "type": enum (Type),
  "format": string,
  "description": string,
  "nullable": boolean,
  "enum": [
    string
  ],
  "maxItems": integer,
  "minItems": integer,
  "properties": {
    string: {
      object (Schema)
    },
    ...
  },
  "required": [
    string
  ],
  "propertyOrdering": [
    string
  ],
  "items": {
    object (Schema)
  }
}
```

The `Type` of the schema must be one of the OpenAPI Data Types (https://spec.openapis.org/oas/v3.0.3#data-types), or a union of those types (using `anyOf`). Only a

subset of fields is valid for each `Type`. The following list maps each `Type` to a subset of the fields that are valid for that type:

- `string` -> `enum`, `format`, `nullable`
- `integer` -> `format`, `minimum`, `maximum`, `enum`, `nullable`
- `number` -> `format`, `minimum`, `maximum`, `enum`, `nullable`
- `boolean` -> `nullable`
- `array` -> `minItems`, `maxItems`, `items`, `nullable`
- `object` -> `properties`, `required`, `propertyOrdering`, `nullable`

Here are some example schemas showing valid type-and-field combinations:

```
{ "type": "string", "enum": ["a", "b", "c"] }

{ "type": "string", "format": "date-time" }

{ "type": "integer", "format": "int64" }

{ "type": "number", "format": "double" }

{ "type": "boolean" }

{ "type": "array", "minItems": 3, "maxItems": 3, "items": { "type": ... } }

{ "type": "object",
  "properties": {
    "a": { "type": ... },
    "b": { "type": ... },
    "c": { "type": ... }
  },
  "nullable": true,
  "required": ["c"],
  "propertyOrdering": ["c", "b", "a"]
}
```

For complete documentation of the Schema fields as they're used in the Gemini API, see the Schema reference (/api/caching#Schema).

Property ordering

Warning: When you're configuring a JSON schema, make sure to set **`propertyOrdering[]`**, and when you provide examples, make sure that the property ordering in the examples matches the schema.

When you're working with JSON schemas in the Gemini API, the order of properties is important. By default, the API orders properties alphabetically and does not preserve the order in which the properties are defined (although the Google Gen AI SDKs (/gemini-api/docs/sdks) may preserve this order). If you're providing examples to the model with a schema configured, and the property ordering of the examples is not consistent with the property ordering of the schema, the output could be rambling or unexpected.

To ensure a consistent, predictable ordering of properties, you can use the optional **`propertyOrdering[]`** field.

```
"propertyOrdering": ["recipeName", "ingredients"]
```

`propertyOrdering[]` – not a standard field in the OpenAPI specification – is an array of strings used to determine the order of properties in the response. By specifying the order of properties and then providing examples with properties in that same order, you can potentially improve the quality of results. **`propertyOrdering`** is only supported when you manually create **`types.Schema`**.

Schemas in Python

When you're using the Python library, the value of **`response_schema`** must be one of the following:

- A type, as you would use in a type annotation (see the Python **`typing`** module (https://docs.python.org/3/library/typing.html))
- An instance of **`genai.types.Schema`** (https://googleapis.github.io/python-genai/genai.html#genai.types.Schema)
- The **`dict`** equivalent of **`genai.types.Schema`**

The easiest way to define a schema is with a Pydantic type (as shown in the previous example):

Python
(#python)

```
config={'response_mime_type': 'application/json',
        'response_schema': list[Recipe]}
```

When you use a Pydantic type, the Python library builds out a JSON schema for you and sends it to the API. For additional examples, see the Python library docs (https://googleapis.github.io/python-genai/index.html#json-response-schema).

The Python library supports schemas defined with the following types (where `AllowedType` is any allowed type):

- `int`
- `float`
- `bool`
- `str`
- `list[AllowedType]`
- `AllowedType|AllowedType|...`
- For structured types:
  - `dict[str, AllowedType]`. This annotation declares all dict values to be the same type, but doesn't specify what keys should be included.
  - User-defined Pydantic models (https://docs.pydantic.dev/latest/concepts/models/). This approach lets you specify the key names and define different types for the values associated with each of the keys, including nested structures.

JSON Schema support

JSON Schema (https://json-schema.org/) is a more recent specification than OpenAPI 3.0, which the Schema (/api/caching#Schema) object is based on. Support for JSON Schema is available as a preview using the field `responseJsonSchema` (/api/generate-content#FIELDS.response_json_schema) which accepts any JSON Schema with the following limitations:

- It only works with Gemini 2.5.
- While all JSON Schema properties can be passed, not all are supported. See the documentation (/api/generate-content#FIELDS.response_json_schema) for the field for more details.
- Recursive references can only be used as the value of a non-required object property.
- Recursive references are unrolled to a finite degree, based on the size of the schema.
- Schemas that contain `$ref` cannot contain any properties other than those starting with a `$`.

Here's an example of generating a JSON Schema with Pydantic and submitting it to the model:

```
curl "https://generativelanguage.googleapis.com/v1alpha/models/\
gemini-2.5-flash:generateContent?key=$GEMINI_API_KEY" \
    -H 'Content-Type: application/json' \
    -d @- <<EOF
{
  "contents": [{
    "parts":[{
      "text": "Please give a random example following this schema"
    }]
  }],
  "generationConfig": {
    "response_mime_type": "application/json",
    "response_json_schema": $(python3 - << PYEOF
from enum import Enum
from typing import List, Optional, Union, Set
from pydantic import BaseModel, Field, ConfigDict
import json

class UserRole(str, Enum):
    ADMIN = "admin"
    VIEWER = "viewer"

class Address(BaseModel):
    street: str
    city: str

class UserProfile(BaseModel):
    username: str = Field(description="User's unique name")
    age: Optional[int] = Field(ge=0, le=120)
```

```
    roles: Set[UserRole] = Field(min_items=1)
    contact: Union[Address, str]
    model_config = ConfigDict(title="User Schema")

# Generate and print the JSON Schema
print(json.dumps(UserProfile.model_json_schema(), indent=2))
PYEOF
)
  }
}
EOF
```

Passing JSON Schema directly is not yet supported when using the SDK.

Best practices

Keep the following considerations and best practices in mind when you're using a response schema:

- The size of your response schema counts towards the input token limit.
- By default, fields are optional, meaning the model can populate the fields or skip them. You can set fields as required to force the model to provide a value. If there's insufficient context in the associated input prompt, the model generates responses mainly based on the data it was trained on.
- A complex schema can result in an `InvalidArgument: 400` error. Complexity might come from long property names, long array length limits, enums with many values, objects with lots of optional properties, or a combination of these factors.

  If you get this error with a valid schema, make one or more of the following changes to resolve the error:

  - Shorten property names or enum names.
  - Flatten nested arrays.
  - Reduce the number of properties with constraints, such as numbers with minimum and maximum limits.
  - Reduce the number of properties with complex constraints, such as properties with complex formats like `date-time`.

    - Reduce the number of optional properties.
    - Reduce the number of valid values for enums.
- If you aren't seeing the results you expect, add more context to your input prompts or revise your response schema. For example, review the model's response without structured output to see how the model responds. You can then update your response schema so that it better fits the model's output.

What's next

Now that you've learned how to generate structured output, you might want to try using Gemini API tools:

- Function calling (/gemini-api/docs/function-calling)
- Code execution (/gemini-api/docs/code-execution)
- Grounding with Google Search (/gemini-api/docs/grounding)

Last updated 2025-06-17 UTC.