Structured Outputs

You can configure Gemini models to generate responses that adhere to a provided JSON Schema. This capability guarantees predictable and parsable results, ensures format and type-safety, enables the programmatic detection of refusals, and simplifies prompting.

Using structured outputs is ideal for a wide range of applications:

- **Data extraction:** Pull specific information from unstructured text, like extracting names, dates, and amounts from an invoice.
- **Structured classification:** Classify text into predefined categories and assign structured labels, such as categorizing customer feedback by sentiment and topic.
- **Agentic workflows:** Generate structured data that can be used to call other tools or APIs, like creating a character sheet for a game or filling out a form.

In addition to supporting JSON Schema in the REST API, the Google GenAI SDKs for Python and JavaScript also make it easy to define object schemas using Pydantic (https://docs.pydantic.dev/latest/) and Zod (https://zod.dev/), respectively. The example below demonstrates how to extract information from unstructured text that conforms to a schema defined in code.

✓ Recipe Extractor Content Moderation Recursive Structures

This example demonstrates how to extract structured data from text using basic JSON Schema types like `object`, `array`, `string`, and `integer`.

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

```
import { GoogleGenAI } from "@google/genai";
import { z } from "zod";
import { zodToJsonSchema } from "zod-to-json-schema";

const ingredientSchema = z.object({
  name: z.string().describe("Name of the ingredient."),
  quantity: z.string().describe("Quantity of the ingredient, including units."),
```

```
});

const recipeSchema = z.object({
  recipe_name: z.string().describe("The name of the recipe."),
  prep_time_minutes: z.number().optional().describe("Optional time in minutes to prepare the recipe."),
  ingredients: z.array(ingredientSchema),
  instructions: z.array(z.string()),
});

const ai = new GoogleGenAI({});

const prompt = `
Please extract the recipe from the following text.
The user wants to make delicious chocolate chip cookies.
They need 2 and 1/4 cups of all-purpose flour, 1 teaspoon of baking soda,
1 teaspoon of salt, 1 cup of unsalted butter (softened), 3/4 cup of granulated sugar,
3/4 cup of packed brown sugar, 1 teaspoon of vanilla extract, and 2 large eggs.
For the best part, they'll need 2 cups of semisweet chocolate chips.
First, preheat the oven to 375°F (190°C). Then, in a small bowl, whisk together the flour,
baking soda, and salt. In a large bowl, cream together the butter, granulated sugar, and brown sugar
until light and fluffy. Beat in the vanilla and eggs, one at a time. Gradually beat in the dry
ingredients until just combined. Finally, stir in the chocolate chips. Drop by rounded tablespoons
onto ungreased baking sheets and bake for 9 to 11 minutes.
`;

const response = await ai.models.generateContent({
  model: "gemini-2.5-flash",
  contents: prompt,
  config: {
    responseMimeType: "application/json",
    responseJsonSchema: zodToJsonSchema(recipeSchema),
  },
});

const recipe = recipeSchema.parse(JSON.parse(response.text));
console.log(recipe);
```

Example Response:

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

```
    }
  ],
  "instructions": [
    "Preheat the oven to 375°F (190°C).",
    "In a small bowl, whisk together the flour, baking soda, and salt.",
    "In a large bowl, cream together the butter, granulated sugar, and brown sugar until light and fluffy.",
    "Beat in the vanilla and eggs, one at a time.",
    "Gradually beat in the dry ingredients until just combined.",
    "Stir in the chocolate chips.",
    "Drop by rounded tablespoons onto ungreased baking sheets and bake for 9 to 11 minutes."
  ]
}
```

Streaming

You can stream structured outputs, which allows you to start processing the response as it's being generated, without having to wait for the entire output to be complete. This can improve the perceived performance of your application.

The streamed chunks will be valid partial JSON strings, which can be concatenated to form the final, complete JSON object.

Python (#python) JavaScript (#javascript)

```
import { GoogleGenAI } from "@google/genai";
import { z } from "zod";
import { zodToJsonSchema } from "zod-to-json-schema";

const ai = new GoogleGenAI({});
const prompt = "The new UI is incredibly intuitive and visually appealing. Great job! Add a very long summary to test streaming!"

const feedbackSchema = z.object({
  sentiment: z.enum(["positive", "neutral", "negative"]),
  summary: z.string(),
});
```

```
const stream = await ai.models.generateContentStream({
  model: "gemini-2.5-flash",
  contents: prompt,
  config: {
    responseMimeType: "application/json",
    responseJsonSchema: zodToJsonSchema(feedbackSchema),
  },
});

for await (const chunk of stream) {
  console.log(chunk.candidates[0].content.parts[0].text)
}
```

Structured outputs with tools

Preview: This is a feature available only for the Gemini 3 series models, **`gemini-3-pro-preview`** and **`gemini-3-flash-preview`**.

Gemini 3 lets you combine Structured Outputs with built-in tools, including Grounding with Google Search (/gemini-api/docs/google-search), URL Context (/gemini-api/docs/url-context), and Code Execution (/gemini-api/docs/code-execution).

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

```
import { GoogleGenAI } from "@google/genai";
import { z } from "zod";
import { zodToJsonSchema } from "zod-to-json-schema";

const ai = new GoogleGenAI({});

const matchSchema = z.object({
  winner: z.string().describe("The name of the winner."),
  final_match_score: z.string().describe("The final score."),
  scorers: z.array(z.string()).describe("The name of the scorer.")
```

```
});

async function run() {
  const response = await ai.models.generateContent({
    model: "gemini-3-pro-preview",
    contents: "Search for all details for the latest Euro.",
    config: {
      tools: [
        { googleSearch: {} },
        { urlContext: {} }
      ],
      responseMimeType: "application/json",
      responseJsonSchema: zodToJsonSchema(matchSchema),
    },
  });

  const match = matchSchema.parse(JSON.parse(response.text));
  console.log(match);
}

run();
```

JSON schema support

To generate a JSON object, set the **`response_mime_type`** in the generation configuration to **`application/json`** and provide a **`response_json_schema`**. The schema must be a valid JSON Schema (https://json-schema.org/) that describes the desired output format.

The model will then generate a response that is a syntactically valid JSON string matching the provided schema. When using structured outputs, the model will produce outputs in the same order as the keys in the schema.

Gemini's structured output mode supports a subset of the JSON Schema (https://json-schema.org) specification.

The following values of **`type`** are supported:

- **`string`**: For text.

- **`number`**: For floating-point numbers.
- **`integer`**: For whole numbers.
- **`boolean`**: For true/false values.
- **`object`**: For structured data with key-value pairs.
- **`array`**: For lists of items.
- **`null`**: To allow a property to be null, include "`null`" in the type array (e.g., `{"type": ["string", "null"]}`).

These descriptive properties help guide the model:

- **`title`**: A short description of a property.
- **`description`**: A longer and more detailed description of a property.

Type-specific properties

For `object` values:

- **`properties`**: An object where each key is a property name and each value is a schema for that property.
- **`required`**: An array of strings, listing which properties are mandatory.
- **`additionalProperties`**: Controls whether properties not listed in `properties` are allowed. Can be a boolean or a schema.

For `string` values:

- **`enum`**: Lists a specific set of possible strings for classification tasks.
- **`format`**: Specifies a syntax for the string, such as `date-time`, `date`, `time`.

For `number` and `integer` values:

- **`enum`**: Lists a specific set of possible numeric values.
- **`minimum`**: The minimum inclusive value.

- **`maximum`**: The maximum inclusive value.

For `array` values:

- **`items`**: Defines the schema for all items in the array.
- **`prefixItems`**: Defines a list of schemas for the first N items, allowing for tuple-like structures.
- **`minItems`**: The minimum number of items in the array.
- **`maxItems`**: The maximum number of items in the array.

Model support

The following models support structured output:

Model	Structured Outputs
Gemini 3 Pro Preview	✔
Gemini 3 Flash Preview	✔
Gemini 2.5 Pro	✔
Gemini 2.5 Flash	✔
Gemini 2.5 Flash-Lite	✔
Gemini 2.0 Flash	✔*
Gemini 2.0 Flash-Lite	✔*

** Note that Gemini 2.0 requires an explicit `propertyOrdering` list within the JSON input to define the preferred structure. You can find an example in this cookbook (https://github.com/google-gemini/cookbook/blob/main/examples/Pdf_structured_outputs_on_invoices_and_forms.ipynb).*

Structured outputs vs. function calling

Both structured outputs and function calling use JSON schemas, but they serve different purposes:

Feature	Primary Use Case
Structured Outputs	**Formatting the final response to the user.** Use this when you want the model's *answer* to be in a specific format (e.g., extracting data from a document to save to a database).
Function Calling	**Taking action during the conversation.** Use this when the model needs to *ask you* to perform a task (e.g., "get current weather") before it can provide a final answer.

Best practices

- **Clear descriptions:** Use the `description` field in your schema to provide clear instructions to the model about what each property represents. This is crucial for guiding the model's output.
- **Strong typing:** Use specific types (`integer`, `string`, `enum`) whenever possible. If a parameter has a limited set of valid values, use an `enum`.
- **Prompt engineering:** Clearly state in your prompt what you want the model to do. For example, "Extract the following information from the text..." or "Classify this feedback according to the provided schema...".
- **Validation:** While structured output guarantees syntactically correct JSON, it does not guarantee the values are semantically correct. Always validate the final output in your application code before using it.
- **Error handling:** Implement robust error handling in your application to gracefully manage cases where the model's output, while schema-compliant, may not meet your business logic requirements.

Limitations

- **Schema subset:** Not all features of the JSON Schema specification are supported. The model ignores unsupported properties.
- **Schema complexity:** The API may reject very large or deeply nested schemas. If you encounter errors, try simplifying your schema by shortening property names, reducing nesting, or limiting the number of constraints.

Last updated 2025-12-18 UTC.