generative-ai-jupyterlab

File Edit View Run Kernel Git Tabs Settings Help

e2-medium

Launcher | intro_function_calling-v2.0.0.i

Code | git | Execute | Python 3 (ipykernel) (Local)

Filter files by name

/

Name

intro_function_calling...
notebook_template.i...
requirements.txt

[ ]:
```
# Copyright 2024 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "license");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Intro to Function Calling with the Gemini API & Python SDK

Run in Colab | Run in Colab Enterprise | View on GitHub | Open in Vertex AI Workbench | Open in Cloud Skills Boost

Share to:

Authors
Kristopher Overholt
Holt Skinner

Overview

YouTube Video: AI + your code: Function Calling

Gemini

Gemini is a family of generative AI models developed by Google DeepMind that is designed for multimodal use cases.

Calling functions from Gemini

Function Calling in Gemini lets developers create a description of a function in their code, then pass that description to a language model in a request. The response from the model includes the name of a function that matches the description and the arguments to call it with.

Why function calling?

Imagine asking someone to write down important information without giving them a form or any guidelines on the structure. You might get a beautifully crafted paragraph, but extracting specific details like names, dates, or numbers would be tedious! Similarly, trying to get consistent structured data from a generative text model without function calling can be frustrating. You're stuck explicitly prompting for things like JSON output, often with inconsistent and frustrating results.

This is where Gemini Function Calling comes in. Instead of hoping for the best in a freeform text response from a generative model, you can define clear functions with specific parameters and data types. These function declarations act as structured guidelines, guiding the Gemini model to structure its output in a predictable and usable way. No more parsing text responses for important information!

Think of it like teaching Gemini to speak the language of your applications. Need to retrieve information from a database? Define a `search_db` function with parameters for search terms. Want to integrate with a weather API? Create a `get_weather` function that takes a location as input. Function calling bridges the gap between human language and the structured data needed to interact with external systems.

Objectives

In this tutorial, you will learn how to use the Gemini API in Vertex AI with the Vertex AI SDK for Python to make function calls via the Gemini 2.0 Flash (`gemini-2.0-flash`) model.

You will complete the following tasks:

- Install the Google Gen AI SDK for Python
- Use the Gemini API in Vertex AI to interact with the Gemini model:
- Use Function Calling in a chat session to answer user's questions about products in the Google Store
- Use Function Calling to geocode addresses with a maps API
- Use Function Calling for entity extraction on raw logging data

Costs

This tutorial uses billable components of Google Cloud:

- Vertex AI

Learn about Vertex AI pricing and use the Pricing Calculator to generate a cost estimate based on your projected usage.

Getting Started

Install Google Gen AI SDK

[8]:
```
%pip install --upgrade --quiet google-genai
```

```
Note: you may need to restart the kernel to use updated packages.
```

Restart current runtime

To use the newly installed packages in this Jupyter runtime, you must restart the runtime. You can do this by running the cell below, which will restart the current kernel.

[9]:
```
# Restart kernel after installs so that your environment can access the new packages
import IPython

app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

[9]:
```
{'status': 'ok', 'restart': True}
```

⚠️ **The kernel is going to restart. Please wait until it is finished before continuing to the next step.** ⚠️

Set Google Cloud project information

To get started using Vertex AI, you must have an existing Google Cloud project and enable the Vertex AI API.

Learn more about setting up a project and a development environment.

Initialize the Gen AI SDK for Python for your project:

[1]:
```
# Define project information
PROJECT_ID = "qwiklabs-gcp-02-7c93534d3d32"  # @param {type:"string"}
LOCATION = "us-east1"  # @param {type:"string"}

# # Create the API client
from google import genai
client = genai.Client(vertexai=True, project=PROJECT_ID, location=LOCATION)
```

Code Examples

Choose a model

For more information about all AI models and APIs on Vertex AI, see Google Models and Model Garden.

[2]:
```
MODEL_ID = "gemini-2.0-flash-001"  # @param {type: "string"}
```

Import libraries

[3]:
```
from IPython.display import Markdown, display
from google.genai.types import FunctionDeclaration, GenerateContentConfig, Part, Tool
import requests
```

Chat example: Using Function Calling in a chat session to answer user's questions about the Google Store

In this example, you'll use Function Calling along with the chat modality in the Gemini model to help customers get information about products in the Google Store.

You'll start by defining three functions: one to get product information, another to get the location of the closest stores, and one more to place an order:

[4]:
```
get_product_info = FunctionDeclaration(
    name="get_product_info",
    description="Get the stock amount and identifier for a given product",
    parameters={
        "type": "OBJECT",
        "properties": {
            "product_name": {"type": "STRING", "description": "Product name"}
        },
    },
)

get_store_location = FunctionDeclaration(
    name="get_store_location",
    description="Get the location of the closest store",
    parameters={
        "type": "OBJECT",
        "properties": {"location": {"type": "STRING", "description": "Location"}},
    },
)

place_order = FunctionDeclaration(
    name="place_order",
    description="Place an order",
    parameters={
        "type": "OBJECT",
        "properties": {
            "product": {"type": "STRING", "description": "Product name"},
            "address": {"type": "STRING", "description": "Shipping address"},
        },
    },
)
```

Note that function parameters are specified as a Python dictionary in accordance with the OpenAPI JSON schema format.

Define a tool that allows the Gemini model to select from the set of 3 functions:

[5]:
```
retail_tool = Tool(
    function_declarations=[
        get_product_info,
        get_store_location,
        place_order,
    ],
)
```

Now you can initialize the Gemini model with Function Calling in a multi-turn chat session.

You can specify the `tools` kwarg when initializing the chat session to avoid having to send it with every subsequent request:

[6]:
```
chat = client.chats.create(
    model=MODEL_ID,
    config=GenerateContentConfig(
        temperature=0,
        tools=[retail_tool],
    ),
)
```

Note: The `temperature` parameter controls the degree of randomness in this generation. Lower temperatures are good for functions that require deterministic parameter values, while higher temperatures are good for functions with parameters that accept more diverse or creative parameter values. A temperature of `0` is deterministic. In this case, responses for a given prompt are mostly deterministic, but a small amount of variation is still possible.

We're ready to chat! Let's start the conversation by asking if a certain product is in stock:

[7]:
```
prompt = """
Do you have the Pixel 9 in stock?
"""

response = chat.send_message(prompt)
response.function_calls[0]
```

[7]:
```
FunctionCall(id=None, args={'product_name': 'Pixel 9'}, name='get_product_info')
```

The response from the Gemini API consists of a structured data object that contains the name and parameters of the function that Gemini selected out of the available functions.

Since this notebook focuses on the ability to extract function parameters and generate function calls, you'll use mock data to feed synthetic responses back to the Gemini model rather than sending a request to an API server (not to worry, we'll make an actual API call in a later example!):

[8]:
```
# Here you can use your preferred method to make an API request and get a response.
# In this example, we'll use synthetic data to simulate a payload from an external API response.

api_response = {"sku": "GA04834-US", "in_stock": "yes"}
```

In reality, you would execute function calls against an external system or database using your desired client library or REST API.

Now, you can pass the response from the (mock) API request and generate a response for the end user:

[9]:
```
response = chat.send_message(
    Part.from_function_response(
        name="get_product_info",
        response={
            "content": api_response,
        },
    ),
)
display(Markdown(response.text))
```

Yes, we have the Pixel 9 in stock. Its SKU is GA04834-US.

Next, the user might ask where they can buy a different phone from a nearby store:

[10]:
```
prompt = """
What about the Pixel 9 Pro XL? Is there a store in
Mountain View, CA that I can visit to try one out?
"""

response = chat.send_message(prompt)
response.function_calls
```

[10]:
```
[FunctionCall(id=None, args={'product_name': 'Pixel 9 Pro XL'}, name='get_product_info'),
 FunctionCall(id=None, args={'location': 'Mountain View, CA'}, name='get_store_location')]
```

Again, you get a response with structured data, but notice that there are two function calls instead of one!

The Gemini model identified that it needs both the `get_product_info` and `get_store_location` functions. Look closely at the prompt that you used in this conversation turn a few cells up, and you'll notice that the user asked about a product -and- the location of a store.

In cases like this when two or more functions are defined (or when the model predicts multiple function calls to the same function), the Gemini model might sometimes return back-to-back or parallel function call responses within a single conversation turn.

This is expected behavior since the Gemini model predicts which functions it should call at runtime, what order it should call dependent functions in, and which function calls can be parallelized, so that the model can gather enough information to generate a natural language response.

Not to worry! You can repeat the same steps as before and build synthetic payloads that would come from an external APIs:

[11]:
```
# Here you can use your preferred method to make an API request and get a response.
# In this example, we'll use synthetic data to simulate a payload from an external API response.

product_info_api_response = {"sku": "GA08475-US", "in_stock": "yes"}
store_location_api_response = {
    "store": "2000 N Shoreline Blvd, Mountain View, CA 94043, US"
}
```

Again, you can pass the responses from the (mock) API requests back to the Gemini model:

[12]:
```
response = chat.send_message(
    [
        Part.from_function_response(
            name="get_product_info",
            response={
                "content": product_info_api_response,
            },
        ),
        Part.from_function_response(
            name="get_store_location",
            response={
                "content": store_location_api_response,
            },
        ),
    ]
)
display(Markdown(response.text))
```

Yes, we have the Pixel 9 Pro XL in stock. Its SKU is GA08475-US. The store located at 2000 N Shoreline Blvd, Mountain View, CA 94043, US is the closest store to you.

Nice work!

Within a single conversation turn, the Gemini model requested 2 function calls in a row before returning a natural language summary. In reality, you might follow this pattern if you need to make an API call to an inventory management system, and another call to a store location database, customer management system, or document repository.

Finally, the user might ask to order a phone and have it shipped to their address:

[13]:
```
prompt = """
I'd like to order a Pixel 9 Pro XL and have it shipped to 1155 Borregas Ave, Sunnyvale, CA 94089.
"""

response = chat.send_message(prompt)
response.function_calls
```

[13]:
```
[FunctionCall(id=None, args={'address': '1155 Borregas Ave, Sunnyvale, CA 94089', 'product': 'Pixel 9 Pro XL'}, name='place_order')]
```

Perfect! The Gemini model extracted the user's selected product and their address. Now you can call an API to place the order:

[14]:
```
# This is where you would make an API request to return the status of their order.
# Use synthetic data to simulate a response payload from an external API.

order_api_response = {
    "payment_status": "paid",
    "order_number": 12345,
    "est_arrival": "2 days",
}
```

And send the payload from the external API call so that the Gemini API returns a natural language summary to the end user.

[15]:
```
response = chat.send_message(
    Part.from_function_response(
        name="place_order",
        response={
            "content": order_api_response,
        },
    ),
)
display(Markdown(response.text))
```

OK. I have placed an order for a Pixel 9 Pro XL to be shipped to 1155 Borregas Ave, Sunnyvale, CA 94089. The order number is 12345 and it should arrive in 2 days. The payment status is paid.

And you're done!

You were able to have a multi-turn conversation with the Gemini model using function calls, handling payloads, and generating natural language summaries that incorporated the information from the external systems.

Address example: Using Automatic Function Calling to geocode addresses with a maps API

In this example, you'll define a function that takes multiple parameters as inputs. Then you'll use automatic function calling in the Gemini API to make a live API call to convert an address to latitude and longitude coordinates.

Start by writing a Python function:

[16]:
```
def get_location(
    amenity: str | None = None,
    street: str | None = None,
    city: str | None = None,
    county: str | None = None,
    state: str | None = None,
    country: str | None = None,
    postalcode: str | None = None,
) -> list[dict]:
    """
    Get latitude and longitude for a given location.

    Args:
        amenity (str | None): Amenity or Point of interest.
        street (str | None): Street name.
        city (str | None): City name.
        county (str | None): County name.
        state (str | None): State name.
        country (str | None): Country name.
        postalcode (str | None): Postal code.

    Returns:
        list[dict]: A list of dictionaries with the latitude and longitude of the given location.
                    Returns an empty list if the location cannot be determined.
    """
    base_url = "https://nominatim.openstreetmap.org/search"
    params = {
        "amenity": amenity,
        "street": street,
        "city": city,
        "county": county,
        "state": state,
        "country": country,
        "postalcode": postalcode,
        "format": "json",
    }
    # Filter out None values from parameters
    params = {k: v for k, v in params.items() if v is not None}

    try:
        response = requests.get(base_url, params=params, headers={"User-Agent": "none"})
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        print(f"Error fetching location data: {e}")
        return []
```

In this example, you're asking the Gemini model to extract components of the address into specific fields within a structured data object. These fields are then passed to the function you defined and the result is returned to Gemini to make a natural language response.

Send a prompt that includes an address, such as:

[17]:
```
prompt = """
I want to get the coordinates for the following address:
1600 Amphitheatre Pkwy, Mountain View, CA 94043
"""

response = client.models.generate_content(
    model=MODEL_ID,
```

```
    contents=prompt,
    config=GenerateContentConfig(tools=[get_location], temperature=0),
)
print(response.text)
```

```
The coordinates for 1600 Amphitheatre Pkwy, Mountain View, CA 94043 are: latitude 37.4224857, longitude -122.0855846.
```

Great work! You were able to define a function that the Gemini model used to extract the relevant parameters from the prompt. Then you made a live API call to obtain the coordinates of the specified location.

Here we used the OpenStreetMap Nominatim API to geocode an address to keep the number of steps in this tutorial to a reasonable number. If you're working with large amounts of address or geolocation data, you can also use the Google Maps Geocoding API, or any mapping service with an API!

Conclusions

You have explored the function calling feature through the Google Gen AI Python SDK.

The next step is to enhance your skills by exploring this documentation page.

[ ]: