intro_function_calling

May 4, 2024

[1]:

1 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

Author(s)	Kristopher Overholt

1.1 Overview

1.1.1 Gemini

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

1.1.2 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.

1.1.3 Why function calling?

When working with generative text models, it can be difficult to coerce generative models to give consistent outputs in a structured format such as JSON. Function Calling in Gemini allows you to overcome this limitation by forcing the model to output structured data in the format and schema that you define.

You can think of Function Calling as a way to get structured output from user prompts and function definitions, use that structured output to make an API request to an external system, then return the function response to the generative model so that it can generate a natural language summary. In other words, function calling in Gemini helps you go from unstructured text in prompt, to a structured data object, and back to natural language again.

1.1.4 Objectives

In this tutorial, you will learn how to use the Vertex AI Gemini API with the Vertex AI SDK for Python to make function calls via the Gemini 1.0 Pro (`gemini-1.0-pro`) model.

You will complete the following tasks:

- Install the Vertex AI SDK for Python
- Use the Vertex AI Gemini API to interact with the Gemini 1.0 Pro (`gemini-1.0-pro`) model:
  - Generate function calls from a text prompt to get the weather for a given location
  - Generate function calls from a text prompt and call an external API to geocode addresses
  - Generate function calls from a chat prompt to help retail users

1.1.5 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.

1.2 Getting Started

1.2.1 Install Vertex AI SDK for Python

[2]:
```
!pip3 install --upgrade --user google-cloud-aiplatform
```

```
Requirement already satisfied: google-cloud-aiplatform in
/home/jupyter/.local/lib/python3.10/site-packages (1.50.0)
Requirement already satisfied: google-api-core!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=
2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1 in
/home/jupyter/.local/lib/python3.10/site-packages (from google-api-core[grpc]!=2
.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1-
>google-cloud-aiplatform) (2.19.0)
Requirement already satisfied: google-auth<3.0.0dev,>=2.14.1 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (2.29.0)
Requirement already satisfied: proto-plus<2.0.0dev,>=1.22.0 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (1.23.0)
```

```
Requirement already satisfied: protobuf!=3.20.0,!=3.20.1,!=4.21.0,!=4.21.1,!=4.2
1.2,!=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.19.5 in
/home/jupyter/.local/lib/python3.10/site-packages (from google-cloud-aiplatform)
(4.25.3)
Requirement already satisfied: packaging>=14.3 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (24.0)
Requirement already satisfied: google-cloud-storage<3.0.0dev,>=1.32.0 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (2.14.0)
Requirement already satisfied: google-cloud-bigquery!=3.20.0,<4.0.0dev,>=1.15.0
in /opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform)
(3.21.0)
Requirement already satisfied: google-cloud-resource-manager<3.0.0dev,>=1.3.3 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (1.12.3)
Requirement already satisfied: shapely<3.0.0dev in
/opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (2.0.4)
Requirement already satisfied: pydantic<3 in /opt/conda/lib/python3.10/site-
packages (from google-cloud-aiplatform) (1.10.15)
Requirement already satisfied: docstring-parser<1 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (0.16)
Requirement already satisfied: googleapis-common-protos<2.0.dev0,>=1.56.2 in
/opt/conda/lib/python3.10/site-packages (from google-api-core!=2.0.*,!=2.1.*,!=2
.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1->google-api-core
[grpc]!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,
>=1.34.1->google-cloud-aiplatform) (1.63.0)
Requirement already satisfied: requests<3.0.0.dev0,>=2.18.0 in
/opt/conda/lib/python3.10/site-packages (from google-api-core!=2.0.*,!=2.1.*,!=2
.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1->google-api-core
[grpc]!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,
>=1.34.1->google-cloud-aiplatform) (2.31.0)
Requirement already satisfied: grpcio<2.0dev,>=1.33.2 in
/opt/conda/lib/python3.10/site-packages (from google-api-core[grpc]!=2.0.*,!=2.1
.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1->google-
cloud-aiplatform) (1.48.1)
Requirement already satisfied: grpcio-status<2.0.dev0,>=1.33.2 in
/opt/conda/lib/python3.10/site-packages (from google-api-core[grpc]!=2.0.*,!=2.1
.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1->google-
cloud-aiplatform) (1.48.1)
Requirement already satisfied: cachetools<6.0,>=2.0.0 in
/opt/conda/lib/python3.10/site-packages (from google-
auth<3.0.0dev,>=2.14.1->google-cloud-aiplatform) (4.2.4)
Requirement already satisfied: pyasn1-modules>=0.2.1 in
/opt/conda/lib/python3.10/site-packages (from google-
auth<3.0.0dev,>=2.14.1->google-cloud-aiplatform) (0.4.0)
Requirement already satisfied: rsa<5,>=3.1.4 in /opt/conda/lib/python3.10/site-
packages (from google-auth<3.0.0dev,>=2.14.1->google-cloud-aiplatform) (4.9)
Requirement already satisfied: google-cloud-core<3.0.0dev,>=1.6.0 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-
bigquery!=3.20.0,<4.0.0dev,>=1.15.0->google-cloud-aiplatform) (2.4.1)
```

```
Requirement already satisfied: google-resumable-media<3.0dev,>=0.6.0 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-
bigquery!=3.20.0,<4.0.0dev,>=1.15.0->google-cloud-aiplatform) (2.7.0)
Requirement already satisfied: python-dateutil<3.0dev,>=2.7.2 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-
bigquery!=3.20.0,<4.0.0dev,>=1.15.0->google-cloud-aiplatform) (2.9.0)
Requirement already satisfied: grpc-google-iam-v1<1.0.0dev,>=0.12.4 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-resource-
manager<3.0.0dev,>=1.3.3->google-cloud-aiplatform) (0.12.7)
Requirement already satisfied: google-crc32c<2.0dev,>=1.0 in
/opt/conda/lib/python3.10/site-packages (from google-cloud-
storage<3.0.0dev,>=1.32.0->google-cloud-aiplatform) (1.5.0)
Requirement already satisfied: typing-extensions>=4.2.0 in
/opt/conda/lib/python3.10/site-packages (from pydantic<3->google-cloud-
aiplatform) (4.11.0)
Requirement already satisfied: numpy<3,>=1.14 in /opt/conda/lib/python3.10/site-
packages (from shapely<3.0.0dev->google-cloud-aiplatform) (1.24.4)
Requirement already satisfied: six>=1.5.2 in /opt/conda/lib/python3.10/site-
packages (from grpcio<2.0dev,>=1.33.2->google-api-core[grpc]!=2.0.*,!=2.1.*,!=2.
2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1->google-cloud-
aiplatform) (1.16.0)
Requirement already satisfied: pyasn1<0.7.0,>=0.4.6 in
/opt/conda/lib/python3.10/site-packages (from pyasn1-modules>=0.2.1->google-
auth<3.0.0dev,>=2.14.1->google-cloud-aiplatform) (0.6.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/opt/conda/lib/python3.10/site-packages (from
requests<3.0.0.dev0,>=2.18.0->google-api-core!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2
.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1->google-api-core[grpc]!=2.0.*,!=
2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1-
>google-cloud-aiplatform) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.10/site-
packages (from requests<3.0.0.dev0,>=2.18.0->google-api-core!=2.0.*,!=2.1.*,!=2.
2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1->google-api-core[
grpc]!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>
=1.34.1->google-cloud-aiplatform) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/opt/conda/lib/python3.10/site-packages (from
requests<3.0.0.dev0,>=2.18.0->google-api-core!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2
.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1->google-api-core[grpc]!=2.0.*,!=
2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1-
>google-cloud-aiplatform) (1.26.18)
Requirement already satisfied: certifi>=2017.4.17 in
/opt/conda/lib/python3.10/site-packages (from
requests<3.0.0.dev0,>=2.18.0->google-api-core!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2
.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1->google-api-core[grpc]!=2.0.*,!=
2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0dev,>=1.34.1-
>google-cloud-aiplatform) (2024.2.2)
```

1.2.2 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.

[3]:
```
# Restart kernel after installs so that your environment can access the new␣
 ↪packages
import IPython

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

[3]:
```
{'status': 'ok', 'restart': True}
```

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

1.2.3 Authenticate your notebook environment (Colab only)

If you are running this notebook on Google Colab, run the following cell to authenticate your environment. This step is not required if you are using Vertex AI Workbench.

[ ]:
```
import sys

if "google.colab" in sys.modules:
    from google.colab import auth

    auth.authenticate_user()
```

1.2.4 Set Google Cloud project information and initialize Vertex AI SDK

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.

[1]:
```
PROJECT_ID = "qwiklabs-gcp-00-2dd21031196b"  # @param {type:"string"}
LOCATION = "europe-west4"  # @param {type:"string"}

import vertexai

vertexai.init(project=PROJECT_ID, location=LOCATION)
```

1.3 Code Examples

1.3.1 Import libraries

[2]:
```
import requests
from vertexai.generative_models import (
    Content,
    FunctionDeclaration,
    GenerationConfig,
    GenerativeModel,
    Part,
    Tool,
)
```

1.3.2 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:

[3]:
```
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:

[4]:
```
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 model to avoid having to send this kwarg with every subsequent request:

[5]:
```
model = GenerativeModel(
    "gemini-1.0-pro-001",
    generation_config=GenerationConfig(temperature=0),
    tools=[retail_tool],
)
chat = model.start_chat()
```

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

[6]:
```
prompt = """
Do you have the Pixel 8 Pro in stock?
"""

response = chat.send_message(prompt)
response.candidates[0].content.parts[0]
```

[6]:
```
function_call {
  name: "get_product_info"
  args {
    fields {
      key: "product_name"
      value {
        string_value: "Pixel 8 Pro"
      }
    }
  }
}
```

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!):

```
[7]: # 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:

```
[8]: response = chat.send_message(
         Part.from_function_response(
             name="get_product_sku",
             response={
                 "content": api_response,
             },
         ),
     )
     response.candidates[0].content.parts[0]
```

```
[8]: text: "Yes, we have the Pixel 8 Pro in stock."
```

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

```
[9]: prompt = """
     What about the Pixel 8? Is there a store in
     Mountain View, CA that I can visit to try one out?
     """

     response = chat.send_message(prompt)
     response.candidates[0].content.parts[0]
```

```
[9]: function_call {
       name: "get_store_location"
       args {
         fields {
           key: "location"
           value {
             string_value: "Mountain View, CA"
           }
         }
       }
     }
```

Again, you get a response with structured data. This time, the Gemini model selected the `get_store_location` function.

Now you can build another synthetic payload that would come from an external API:

[10]:
```
# 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 = {"store": "2000 N Shoreline Blvd, Mountain View, CA 94043, US"}
```

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

[11]:
```
response = chat.send_message(
    Part.from_function_response(
        name="get_store_location",
        response={
            "content": api_response,
        },
    ),
)
response.candidates[0].content.parts[0]
```

[11]:
```
function_call {
  name: "get_product_info"
  args {
    fields {
      key: "product_name"
      value {
        string_value: "Pixel 8"
      }
    }
  }
}
```

Wait a minute! Why did the Gemini API respond with a second function call rather than a natural language summary? 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, the Gemini model might sometimes return back-to-back function call responses within a single conversation turn. This is expected behavior since the Gemini model predicts which functions it should call at runtime so that it can gather enough information to generate a natural language response.

Not to worry, you can repeat the same steps as before and build another synthetic payload that would come from an external API:

[12]:
```
# 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": "GA08475-US", "in_stock": "yes"}
```

And you can pass the response from the (mock) API request back to the Gemini model:

[13]:
```
response = chat.send_message(
    Part.from_function_response(
        name="get_product_info",
        response={
            "content": api_response,
        },
    ),
)
response.candidates[0].content.parts[0]
```

[13]:
```
text: "Yes, we have the Pixel 8 in stock. There is a store in Mountain View, CA
at 2000 N Shoreline Blvd, Mountain View, CA 94043, US where you can try one
out."
```

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:

[14]:
```
prompt = """
I'd like to order a Pixel 8 Pro and have it shipped to 1155 Borregas Ave,␣
 ↪Sunnyvale, CA 94089.
"""

response = chat.send_message(prompt)
response.candidates[0].content.parts[0]
```

[14]:
```
function_call {
  name: "place_order"
  args {
    fields {
      key: "product"
      value {
        string_value: "Pixel 8 Pro"
      }
    }
    fields {
      key: "address"
      value {
        string_value: "1155 Borregas Ave, Sunnyvale, CA 94089"
      }
    }
  }
}
```

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

[15]:
```
# 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.

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.

[16]:
```
response = chat.send_message(
    Part.from_function_response(
        name="place_order",
        response={
            "content": api_response,
        },
    ),
)
response.candidates[0].content.parts[0]
```

[16]:
```
text: "OK. I have placed an order for a Pixel 8 Pro and it will be shipped to
1155 Borregas Ave, Sunnyvale, CA 94089. You can expect delivery in 2 days. Your
order number is 12345."
```

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.

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

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

Start by defining a function declaration and wrapping it in a tool:

[17]:
```
get_location = FunctionDeclaration(
    name="get_location",
    description="Get latitude and longitude for a given location",
    parameters={
        "type": "object",
        "properties": {
            "poi": {"type": "string", "description": "Point of interest"},
            "street": {"type": "string", "description": "Street name"},
            "city": {"type": "string", "description": "City name"},
            "county": {"type": "string", "description": "County name"},
            "state": {"type": "string", "description": "State name"},
            "country": {"type": "string", "description": "Country name"},
            "postal_code": {"type": "string", "description": "Postal code"},
        },
    },
)

location_tool = Tool(
    function_declarations=[get_location],
)
```

In this example, you're asking the Gemini model to extract components of the address into specific fields within a structured data object. You can then map this data to specific input fields to use with your REST API or client library.

Send a prompt that includes an address, such as:

[18]:
```
prompt = """
I want to get the coordinates for the following address:
1600 Amphitheatre Pkwy, Mountain View, CA 94043, US
"""

response = model.generate_content(
    prompt,
    generation_config=GenerationConfig(temperature=0),
    tools=[location_tool],
)
response.candidates[0].content.parts[0]
```

[18]:
```
function_call {
  name: "get_location"
  args {
    fields {
      key: "street"
      value {
        string_value: "1600 Amphitheatre Pkwy"
      }
    }
```

```
      fields {
        key: "state"
        value {
          string_value: "CA"
        }
      }
      fields {
        key: "postal_code"
        value {
          string_value: "94043"
        }
      }
      fields {
        key: "country"
        value {
          string_value: "US"
        }
      }
      fields {
        key: "city"
        value {
          string_value: "Mountain View"
        }
      }
    }
  }
```

Now you can reference the parameters from the function call and make a live API request:

[19]:
```python
x = response.candidates[0].content.parts[0].function_call.args

url = "https://nominatim.openstreetmap.org/search?"
for i in x:
    url += '{}="{}"&'.format(i, x[i])
url += "format=json"

headers = {
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/
 ↪537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36"
}
x = requests.get(url, headers=headers)
content = x.json()
content

# Note: if you get a JSONDecodeError when running this cell, try modifying the
# user agent string in the `headers=` line of code in this cell and re-run.
```

[19]:
```
[{'place_id': 377394982,
  'licence': 'Data © OpenStreetMap contributors, ODbL 1.0.
http://osm.org/copyright',
  'osm_type': 'node',
  'osm_id': 2192620021,
  'lat': '37.4217636',
  'lon': '-122.084614',
  'class': 'office',
  'type': 'it',
  'place_rank': 30,
  'importance': 0.6949356759210291,
  'addresstype': 'office',
  'name': 'Google Headquarters',
  'display_name': 'Google Headquarters, 1600, Amphitheatre Parkway, Mountain
View, Santa Clara County, California, 94043, United States',
  'boundingbox': ['37.4217136', '37.4218136', '-122.0846640', '-122.0845640']}]
```

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!

1.3.4 Logging example: Using Function Calling for entity extraction only

In the previous examples, we made use of the entity extraction functionality within Gemini Function Calling so that we could pass the resulting parameters to a REST API or client library. However, you might want to only perform the entity extraction step with Gemini Function Calling and stop there without actually calling an API. You can think of this functionality as a convenient way to transform unstructured text data into structured fields.

In this example, you'll build a log extractor that takes raw log data and transforms it into structured data with details about error messages.

You'll start by specifying a function declaration that represents the schema of the Function Call:

[20]:
```python
extract_log_data = FunctionDeclaration(
    name="extract_log_data",
    description="Extract details from error messages in raw log data",
    parameters={
        "type": "object",
        "properties": {
            "locations": {
                "type": "array",
                "description": "Errors",
                "items": {
                    "description": "Details of the error",
                    "type": "object",
                    "properties": {
                        "error_message": {
                            "type": "string",
                            "description": "Full error message",
                        },
                        "error_code": {"type": "string", "description": "Error␣
 ↪code"},
                        "error_type": {"type": "string", "description": "Error␣
 ↪type"},
                    },
                },
            }
        },
    },
)
```

You can then define a tool for the generative model to call that includes the `extract_log_data`:

Define a tool for the Gemini model to use that includes the log extractor function:

[21]:
```
extraction_tool = Tool(
    function_declarations=[extract_log_data],
)
```

You can then pass the sample log data to the Gemini model. The model will call the log extractor function, and the model output will be a Function Call response.

[22]:
```
prompt = """
[15:43:28] ERROR: Could not process image upload: Unsupported file format.␣
 ↪(Error Code: 308)
[15:44:10] INFO: Search index updated successfully.
[15:45:02] ERROR: Service dependency unavailable (payment gateway). Retrying...␣
 ↪(Error Code: 5522)
[15:45:33] ERROR: Application crashed due to out-of-memory exception. (Error␣
 ↪Code: 9001)
"""

response = model.generate_content(
    prompt,
    generation_config=GenerationConfig(temperature=0),
    tools=[extraction_tool],
)

response.candidates[0].content.parts[0].function_call
```

[22]:
```
name: "extract_log_data"
args {
  fields {
    key: "locations"
    value {
      list_value {
        values {
          struct_value {
            fields {
              key: "error_type"
              value {
                string_value: "ERROR"
              }
            }
            fields {
              key: "error_message"
              value {
                string_value: "Could not process image upload: Unsupported file
format."
              }
            }
            fields {
              key: "error_code"
              value {
                string_value: "308"
              }
            }
          }
        }
        values {
          struct_value {
            fields {
              key: "error_type"
              value {
                string_value: "ERROR"
              }
            }
            fields {
              key: "error_message"
              value {
                string_value: "Service dependency unavailable (payment gateway).
Retrying…"
              }
            }
            fields {
              key: "error_code"
              value {
                string_value: "5522"
              }
            }
          }
        }
        values {
          struct_value {
            fields {
              key: "error_type"
              value {
                string_value: "ERROR"
              }
            }
            fields {
              key: "error_message"
              value {
                string_value: "Application crashed due to out-of-memory
exception."
              }
            }
            fields {
              key: "error_code"
              value {
                string_value: "9001"
              }
            }
          }
        }
      }
    }
  }
}
```

The response includes a structured data object that contains the details of the error messages that appear in the log.

[ ]:

```

```