Quick tip: Review the prerequisites before you run the lab

End Lab 00:27:58

Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. Learn more.

Open Google Cloud Console

Username

student-00-bd65c460fea4

Password

IFxtnbV9eIil

GCP Project ID

qwiklabs-gcp-04-a4e8d50

Introduction to Function Calling with Gemini

Lab 1 hour No cost Intermediate

★★★★½

This lab may incorporate AI tools to support your learning.

Gemini

Gemini is a family of generative AI models developed by Google DeepMind that is designed for multimodal use cases. The Gemini API gives you access to the Gemini Pro Vision and Gemini Pro models.

Previous Next >

The Vertex AI Gemini API provides a unified interface for interacting with Gemini models. There are currently two models available in the Gemini API:

Function calling 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.

Function calling is similar to Vertex AI Extensions in that they both generate information about functions. The difference between them is that function calling returns JSON data with the name of a function and the arguments to use in your code, whereas Vertex AI Extensions returns the function and calls it for you.

Objectives

In this lab, 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 Pro (`gemini-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 Pro (gemini-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.

Setup

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This Qwiklabs hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

What you need

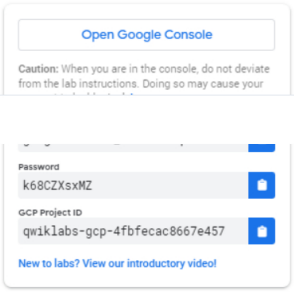
- Access to a standard internet browser (Chrome browser recommended).
- Time to complete the lab.

Note: If you already have your own personal Google Cloud account or project, do not use it for this lab.

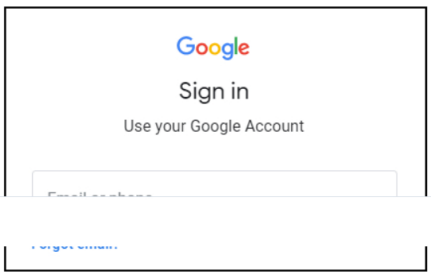
Note: If you are using a Pixelbook, open an Incognito window to run this lab.

How to start your lab and sign in to the Google Cloud Console

1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.

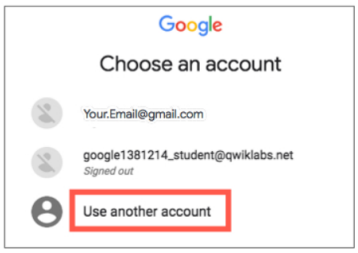


2. Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.



Tip: Open the tabs in separate windows, side-by-side.

If you see the **Choose an account** page, click **Use Another Account**.



3. In the **Sign in** page, paste the username that you copied from the Connection Details

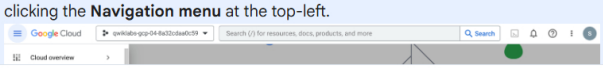
Important: You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

4. Click through the subsequent pages:

- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.

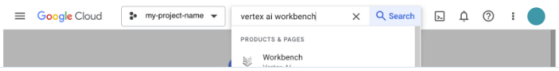
After a few moments, the Cloud Console opens in this tab.

Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.



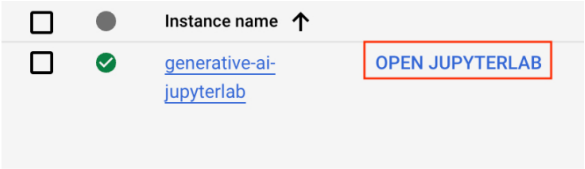
Task 1. Open Python Notebook and Install Packages

1. In the Google Cloud console, navigate to **Vertex AI Workbench**. In the top search bar of the Google Cloud console, enter **Vertex AI Workbench**, and click on the first result.



2. Under **Instances**, click on **Open JupyterLab** for **generative-ai-jupyterlab** instance.

   The JupyterLab will run in a new tab.



3. On the **Launcher**, under **Notebook**, click on **Python 3** to open a new python notebook.

4. Install Vertex AI SDK for Python by the running the following command in the first cell of the notebook. Either click the play ▶ button at the top or enter

```
! pip3 install --upgrade --user google-cloud-aiplatform
```

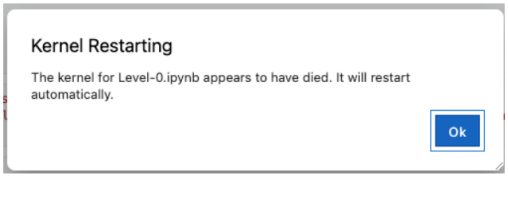
Output:

5. To use the newly installed packages in this Jupyter runtime, it is recommended to restart the runtime. Restart the kernel by running the below code snippet or clicking the refresh button ↻ at the top, followed by clicking **Restart** button.

```
import IPython

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

Output:



After the restart is complete, click **Ok** on the prompt to continue.

```
import requests
from vertexai.preview.generative_models import (
    Content,
    FunctionDeclaration,
    GenerativeModel,
    Part,
    Tool,
)
```

Task 2. Simple Function Calling

Why function calling?

When working with a generative text model, it can be difficult to coerce the LLM to give consistent responses in a structured format such as JSON. Function calling makes it easy to work with LLMs via prompts and unstructured inputs, and have the LLM return a structured response that can be used to call an external function.

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 LLM to generate a response to the user. In other words, function calling in Gemini extracts structured parameters from unstructured text or messages from users.

Use the Gemini Pro model

The Gemini Pro (`gemini-pro`) model is designed to handle natural language tasks,

1. Load the Gemini Pro model:

```
model = GenerativeModel("gemini-pro")
```

Simple function calling

We'll use function calling to set up a weather API request for users to obtain the current conditions in a given location. Function parameters are specified as a Python dictionary in accordance with the OpenAPI JSON schema format.

1. Run the following code snippet to specify function declaration and parameters needed to make a request for our weather API.

```
    description="Get the current weather in a given location",
    parameters={
    "type": "object",
    "properties": {
        "location": {
            "type": "string",
            "description": "Location"
        }
    }
},
)
```

2. Run the following code snippet to define a tool for the LLM to call that includes the `get_current_weather_func`.

```
weather_tool = Tool(
```

```
    function_declarations=[get_current_weather_func],
```

3. We will then instruct the model to generate content. Include the `tool` that you just created to generate a response:

```
prompt = "What is the weather like in Boston?"

response = model.generate_content(
    prompt,
    generation_config={"temperature": 0},
    tools=[weather_tool],
)
response
```

Output:

```
candidates {
  content {
    function_call {
      name: "get_current_weather"
      args {
        fields {
          key: "location"
          value {
            string_value: "Boston"
          }
        }
      }
    }
  }
  finish_reason: STOP
  safety_ratings {
    category: HARM_CATEGORY_HARASSMENT
    probability: NEGLIGIBLE
  }
  safety_ratings {
    category: HARM_CATEGORY_SEXUALLY_EXPLICIT
    probability: NEGLIGIBLE
  }
  safety_ratings {
    category: HARM_CATEGORY_DANGEROUS_CONTENT
    probability: NEGLIGIBLE
  }
}
usage_metadata {
  prompt_token_count: 8
  total_token_count: 8
}
```

4. Let's inspect the function call portion of the response.

Output:

```
name: "get_current_weather"
args {
  fields {
    key: "location"
    value {
      string_value: "Boston"
    }
  }
}
```

The generated response includes a function signature that can be used to call the weather API. Now, we have everything that we need to form a request body and make an API call to an external system.

Task 3. Complex Function Calling

Let's generate a function call that has a more complex structure. Let's use the function response to make an API call that converts an address to latitude and longitude coordinates.

1. Run the following code snippet to define a function declaration within a tool:

```
get_location = FunctionDeclaration(
    name="get_location",
    description="Get latitude and longitude for a given
location",
    "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",
        "country": {
            "type": "string",
            "description": "Country name"
        },
        "postal_code": {
            "type": "string",
            "description": "Postal code"
        },
    },
},
)
location_tool = Tool(
    function_declarations=[get_location],
)
```

2. Next, call the model to generate a response.

```
1600 Amphitheatre Pkwy, Mountain View, CA 94043, US
"""
response = model.generate_content(
    prompt,
    generation_config={"temperature": 0},
    tools=[location_tool],
)
response.candidates[0].content.parts[0]
```

Output:

```
function_call {
  name: "get_location"
  args {
    fields {
      key: "city"
      value {
    }
```

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

```
      key: "street"
      value {
        string_value: "1600 Amphitheatre Pkwy"
      }
    }
  }
}
```

3. Then, extract the results from the function response and make an API call.

```
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 = {
```

```
Safari/537.36
}
x = requests.get(url, headers=headers)
content = x.json()
content
```

Output:

```
[{'place_id': 377680635,
  'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/
  'osm_type': 'node',
  'osm_id': 2192620021,
  'lat': '37.4217636',
  'lon': '-122.084614',
  'class': 'office',
  'type': 'it',
  'place_rank': 30,
```

```
  'name': 'Google Headquarters',
  'display_name': 'Google Headquarters, 1600, Amphitheatre Parkway, Mounta
  'boundingbox': ['37.4217136', '37.4218136', '-122.0846640', '-122.084564
```

Great work! You were able to construct a function and tool that the LLM used to output the parameters necessary for a function call, then you actually made the function call to obtain the coordinates of the specified location.

Here we used the OpenStreetMap Nominatim API to geocode an address to make it easy to use and learn in this notebook. If you're working with large amounts of maps or geolocation data, you can use the Google Maps Geocoding API.

Next, let's use the chat model in Gemini to help customers get information about products in a store.

products in a store.

1. Let's start by defining multiple functions within a tool for getting the product information, the location of stores and placing an order.

```
get_product_info_func = FunctionDeclaration(
    name="get_product_sku",
    description="Get the SKU for a product",
    parameters={
    "type": "object",
    "properties": {
        "product_name": {
            "type": "string",
            "description": "Product name"
        }
```

```
)
get_store_location_func = FunctionDeclaration(
    name="get_store_location",
    description="Get the location of the closest store",
    parameters={
    "type": "object",
    "properties": {
        "location": {
            "type": "string",
            "description": "Location"
        }
    }
},
)
place_order_func = FunctionDeclaration(
    name="place_order",
    description="Place an order",
    parameters={
    "type": "object",
```

```
            "type": "string",
            "description": "Product name"
        },
        "account": {
            "type": "integer",
            "description": "Account number"
        },
        "address": {
            "type": "string",
            "description": "Shipping address"
        }
    }
},
)
retail_tool = Tool(
    function_declarations=[get_product_info_func,
                           get_store_location_func,
                           place_order_func,
```

2. Function calling can also be used in a multi-turn chat session. You can specify tools when creating a model to avoid having to send them with every request.

```
model = GenerativeModel("gemini-pro",
                        generation_config={"temperature": 0},
                        tools=[retail_tool])
chat = model.start_chat()
```

3. Start the conversation by asking if they have a certain product in stock.

```
prompt = """
Do you have the Pixel 8 Pro in stock?
"""

response = chat.send_message(prompt)
```

Output:

```
function_call {
```

```
  name: "get_product_sku"
  args {
    fields {
      key: "product_name"
      value {
        string_value: "Pixel 8 Pro"
      }
    }
  }
}
```

As expected, the response includes a structured function call that we can use to communicate with external systems.

Since this lab focuses on the ability to extract function parameters and generate function calls, you'll use mock data to feed responses back to the model rather than using an actual API server.

4. Use synthetic data to simulate a response payload from an external API.

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

5. Let's include details from the external API call and generate a response to the user.

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

Output:

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

6. The user might ask where they can buy a phone from a nearby store.

```
prompt = """
Where can I buy one near Mountain View, CA?
"""

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

Output:

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

We get a response with another structured function call, this time it's set up to use a different function from our tool.

```
api_response = {"store": "1600 Amphitheatre Pkwy, Mountain View, CA 94043, US"}
```

8. Again, let's include details from the external API call and generate a response to the user.

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

```
text: " There is a store at 1600 Amphitheatre Pkwy, Mountain View, CA 9404
```

9. Finally, the user might ask to order a phone and have it shipped to an address.

```
prompt = """
I'd like to place an order for a Pixel 8 Pro and have it shipped
to 1155 Borregas Ave, Sunnyvale, CA 94089, using account number
101.
"""

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

Output:

```
function_call {
  name: "place_order"
  args {
    fields {
      key: "account"
      value {
        string_value: "1234567890"
      }
    }
    fields {
      key: "address"
      value {
        string_value: "1155 Borregas Ave, Sunnyvale, CA 94089"
      }
    }
    fields {
      key: "product"
      value {
        string_value: "Pixel 8 Pro"
      }
    }
  }
}
```

Perfect! We extracted the user's desired product, address, fetched their account number, and now we can call an API to place the order:

10. Use synthetic data to simulate a response payload from an external API.

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

11. Let's include details from the external API call and generate a response to the user.

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

Output:

```
text: " OK. Your order has been placed and will arrive in 2 days. Your ord
```

Congratulations!

You successfully had a multi-turn conversation with Gemini, generated function calls, handled passing (mock) data back to the model, and generated messages that made use of the function responses.

Manual Last Updated October 15, 2024

Lab Last Tested October 15, 2024

Challenge Lab