Install Packages

[1]:
```
! pip3 install --upgrade --user google-cloud-aiplatform
```

```
Requirement already satisfied: google-cloud-aiplatform in ./.local/lib/python3.10/site-packages (1.96.0)
Requirement already satisfied: google-api-core!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0,>=1.34.1 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.*,<3.0.0,>=1.34.1->google-cloud-aiplatform) (2.24.2)
Requirement already satisfied: google-auth<3.0.0,>=2.14.1 in /opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (2.38.0)
Requirement already satisfied: proto-plus<2.0.0,>=1.22.3 in /opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (1.26.1)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<7.0.0,>=3.20.2 in /opt/conda/lib/python3.10/site-packages (fr
om google-cloud-aiplatform) (3.20.3)
Requirement already satisfied: packaging>=14.3 in /opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (24.2)
Requirement already satisfied: google-cloud-storage<3.0.0,>=1.32.0 in /opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (2.19.0)
Requirement already satisfied: google-cloud-bigquery!=3.20.0,<4.0.0,>=1.15.0 in ./.local/lib/python3.10/site-packages (from google-cloud-aiplatform) (3.34.
0)
Requirement already satisfied: google-cloud-resource-manager<3.0.0,>=1.3.3 in /opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (1.14.
2)
Requirement already satisfied: shapely<3.0.0 in /opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (2.0.7)
Requirement already satisfied: google-genai<2.0.0,>=1.0.0 in ./.local/lib/python3.10/site-packages (from google-cloud-aiplatform) (1.18.0)
Requirement already satisfied: pydantic<3 in /opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (2.11.0)
Requirement already satisfied: typing-extensions in /opt/conda/lib/python3.10/site-packages (from google-cloud-aiplatform) (4.13.0)
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<2.0.0,>=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.*,<3.0.0,>=1.34.1->google-api-core[grpc]!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0,>=
1.34.1->google-cloud-aiplatform) (1.69.2)
Requirement already satisfied: requests<3.0.0,>=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.*,<3.0.0,>=1.34.1->google-api-core[grpc]!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0,>=1.34.1->google-c
loud-aiplatform) (2.32.3)
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.*,<3.0.0,>=1.34.1->google-cloud-aiplatform) (1.71.0)
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.*,<3.0.0,>=1.34.1->google-cloud-aiplatform) (1.49.0rc1)
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /opt/conda/lib/python3.10/site-packages (from google-auth<3.0.0,>=2.14.1->google-cloud-aiplatform)
(5.5.2)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /opt/conda/lib/python3.10/site-packages (from google-auth<3.0.0,>=2.14.1->google-cloud-aiplatform)
(0.4.2)
Requirement already satisfied: rsa<5,>=3.1.4 in /opt/conda/lib/python3.10/site-packages (from google-auth<3.0.0,>=2.14.1->google-cloud-aiplatform) (4.9)
Requirement already satisfied: google-cloud-core<3.0.0,>=2.4.1 in /opt/conda/lib/python3.10/site-packages (from google-cloud-bigquery!=3.20.0,<4.0.0,>=1.1
5.0->google-cloud-aiplatform) (2.4.3)
Requirement already satisfied: google-resumable-media<3.0.0,>=2.0.0 in /opt/conda/lib/python3.10/site-packages (from google-cloud-bigquery!=3.20.0,<4.0.0,>
=1.15.0->google-cloud-aiplatform) (2.7.2)
Requirement already satisfied: python-dateutil<3.0.0,>=2.8.2 in /opt/conda/lib/python3.10/site-packages (from google-cloud-bigquery!=3.20.0,<4.0.0,>=1.15.0
->google-cloud-aiplatform) (2.9.0.post0)
Requirement already satisfied: grpc-google-iam-v1<1.0.0,>=0.14.0 in /opt/conda/lib/python3.10/site-packages (from google-cloud-resource-manager<3.0.0,>=1.
3.3->google-cloud-aiplatform) (0.14.2)
Requirement already satisfied: google-crc32c<2.0dev,>=1.0 in /opt/conda/lib/python3.10/site-packages (from google-cloud-storage<3.0.0,>=1.32.0->google-clou
d-aiplatform) (1.7.1)
Requirement already satisfied: anyio<5.0.0,>=4.8.0 in /opt/conda/lib/python3.10/site-packages (from google-genai<2.0.0,>=1.0.0->google-cloud-aiplatform)
(4.9.0)
Requirement already satisfied: httpx<1.0.0,>=0.28.1 in ./.local/lib/python3.10/site-packages (from google-genai<2.0.0,>=1.0.0->google-cloud-aiplatform) (0.
28.1)
Requirement already satisfied: websockets<15.1.0,>=13.0.0 in /opt/conda/lib/python3.10/site-packages (from google-genai<2.0.0,>=1.0.0->google-cloud-aiplatf
orm) (15.0.1)
Requirement already satisfied: annotated-types>=0.6.0 in /opt/conda/lib/python3.10/site-packages (from pydantic<3->google-cloud-aiplatform) (0.7.0)
Requirement already satisfied: pydantic-core==2.33.0 in /opt/conda/lib/python3.10/site-packages (from pydantic<3->google-cloud-aiplatform) (2.33.0)
Requirement already satisfied: typing-inspection>=0.4.0 in /opt/conda/lib/python3.10/site-packages (from pydantic<3->google-cloud-aiplatform) (0.4.0)
Requirement already satisfied: numpy<3,>=1.14 in /opt/conda/lib/python3.10/site-packages (from shapely<3.0.0->google-cloud-aiplatform) (2.1.3)
Requirement already satisfied: exceptiongroup>=1.0.2 in /opt/conda/lib/python3.10/site-packages (from anyio<5.0.0,>=4.8.0->google-genai<2.0.0,>=1.0.0->goog
le-cloud-aiplatform) (1.2.2)
Requirement already satisfied: idna>=2.8 in /opt/conda/lib/python3.10/site-packages (from anyio<5.0.0,>=4.8.0->google-genai<2.0.0,>=1.0.0->google-cloud-aip
latform) (3.10)
Requirement already satisfied: sniffio>=1.1 in /opt/conda/lib/python3.10/site-packages (from anyio<5.0.0,>=4.8.0->google-genai<2.0.0,>=1.0.0->google-cloud-
aiplatform) (1.3.1)
Requirement already satisfied: certifi in /opt/conda/lib/python3.10/site-packages (from httpx<1.0.0,>=0.28.1->google-genai<2.0.0,>=1.0.0->google-cloud-aipl
atform) (2025.1.31)
Requirement already satisfied: httpcore==1.* in ./.local/lib/python3.10/site-packages (from httpx<1.0.0,>=0.28.1->google-genai<2.0.0,>=1.0.0->google-cloud-
aiplatform) (1.0.9)
Requirement already satisfied: h11>=0.16 in ./.local/lib/python3.10/site-packages (from httpcore==1.*->httpx<1.0.0,>=0.28.1->google-genai<2.0.0,>=1.0.0->go
ogle-cloud-aiplatform) (0.16.0)
Requirement already satisfied: pyasn1<0.7.0,>=0.6.1 in /opt/conda/lib/python3.10/site-packages (from pyasn1-modules>=0.2.1->google-auth<3.0.0,>=2.14.1->goo
gle-cloud-aiplatform) (0.6.1)
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.10/site-packages (from python-dateutil<3.0.0,>=2.8.2->google-cloud-bigquery!=3.20.0,<4.0.
0,>=1.15.0->google-cloud-aiplatform) (1.17.0)
Requirement already satisfied: charset_normalizer<4,>=2 in /opt/conda/lib/python3.10/site-packages (from requests<3.0.0,>=2.18.0->google-api-core!=2.0.*,!=
2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0,>=1.34.1->google-api-core[grpc]!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.
*,<3.0.0,>=1.34.1->google-cloud-aiplatform) (3.4.1)
Requirement already satisfied: urllib3<3,>=1.21.1 in /opt/conda/lib/python3.10/site-packages (from requests<3.0.0,>=2.18.0->google-api-core!=2.0.*,!=2.1.
*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.0.0,>=1.34.1->google-api-core[grpc]!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!=2.5.*,!=2.6.*,!=2.7.*,<3.
0.0,>=1.34.1->google-cloud-aiplatform) (1.26.20)
```

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 restart kernel at the top, followed by clicking Restart button.

[2]:
```
import IPython

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

[2]:
```
{'status': 'ok', 'restart': True}
```

Run the following code snippet in the next cell to import the libraries.

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

Simple Function Calling

Why 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, multiturn text and code chat, and code generation.

Load the Gemini Pro model:

[7]:
```
model = GenerativeModel("gemini-2.0-flash-001")
```

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.

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

[8]:
```
get_current_weather_func = FunctionDeclaration(
    name="get_current_weather",
    description="Get the current weather in a given location",
    parameters={
    "type": "object",
    "properties": {
        "location": {
            "type": "string",
            "description": "Location"
        }
    }
},
)
```

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

[9]:
```
weather_tool = Tool(
    function_declarations=[get_current_weather_func],
)
```

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

[10]:
```
prompt = "What is the weather like in Boston?"

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

[10]:
```
candidates {
  content {
    role: "model"
    parts {
      function_call {
        name: "get_current_weather"
        args {
          fields {
            key: "location"
            value {
              string_value: "Boston"
            }
          }
        }
      }
    }
  }
  finish_reason: STOP
  avg_logprobs: -2.4811893776391765e-06
}
usage_metadata {
  prompt_token_count: 25
  candidates_token_count: 7
  total_token_count: 32
  prompt_tokens_details {
    modality: TEXT
    token_count: 25
  }
  candidates_tokens_details {
    modality: TEXT
    token_count: 7
  }
}
model_version: "gemini-2.0-flash-001"
create_time {
  seconds: 1749014306
  nanos: 370469000
}
response_id: "Itc_aKXOFv2fmecPlYPsgQI"
```

Let's inspect the function call portion of the response

[11]:
```
response.candidates[0].content.parts[0].function_call
```

[11]:
```
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.

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.

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

[12]:
```
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],
)
```

Next, call the model to generate a response.

[13]:
```
prompt = """
I want to get the lat/lon coordinates for the following address:
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]
```

[13]:
```
function_call {
  name: "get_location"
  args {
    fields {
      key: "city"
      value {
        string_value: "Mountain View"
      }
    }
    fields {
      key: "country"
      value {
        string_value: "US"
      }
    }
    fields {
      key: "postal_code"
      value {
        string_value: "94043"
      }
    }
    fields {
      key: "state"
      value {
        string_value: "CA"
      }
    }
    fields {
      key: "street"
      value {
        string_value: "1600 Amphitheatre Pkwy"
      }
    }
  }
}
```

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

[21]:
```
import json
```

[23]:
```
x = response.candidates[0].content.parts[0].function_call.args
# print(x)
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)
# print(x.status_code)
# print(x.text)
content = x.json()
content
```

```
---------------------------------------------------------------------------
JSONDecodeError                           Traceback (most recent call last)
```

```
File /opt/conda/lib/python3.10/site-packages/requests/models.py:974, in Response.json(self, **kwargs)
    973 try:
--> 974     return complexjson.loads(self.text, **kwargs)
    975 except JSONDecodeError as e:
    976     # Catch JSON-related errors and raise as requests.JSONDecodeError
    977     # This aliases json.JSONDecodeError and simplejson.JSONDecodeError

File /opt/conda/lib/python3.10/json/__init__.py:346, in loads(s, cls, object_hook, parse_float, parse_int, parse_constant, object_pairs_hook, **kw)
    343 if (cls is None and object_hook is None and
    344         parse_int is None and parse_float is None and
    345         parse_constant is None and object_pairs_hook is None and not kw):
--> 346     return _default_decoder.decode(s)
    347 if cls is None:

File /opt/conda/lib/python3.10/json/decoder.py:337, in JSONDecoder.decode(self, s, _w)
    333 """Return the Python representation of ``s`` (a ``str`` instance
    334 containing a JSON document).
    335
    336 """
--> 337 obj, end = self.raw_decode(s, idx=_w(s, 0).end())
    338 end = _w(s, end).end()

File /opt/conda/lib/python3.10/json/decoder.py:355, in JSONDecoder.raw_decode(self, s, idx)
    354 except StopIteration as err:
--> 355     raise JSONDecodeError("Expecting value", s, err.value) from None
    356 return obj, end

JSONDecodeError: Expecting value: line 1 column 1 (char 0)

During handling of the above exception, another exception occurred:

JSONDecodeError                           Traceback (most recent call last)
Cell In[23], line 13
     10 x = requests.get(url, headers=headers)
     11 # print(x.status_code)
     12 # print(x.text)
---> 13 content = x.json()
     14 content

File /opt/conda/lib/python3.10/site-packages/requests/models.py:978, in Response.json(self, **kwargs)
    974     return complexjson.loads(self.text, **kwargs)
    975 except JSONDecodeError as e:
    976     # Catch JSON-related errors and raise as requests.JSONDecodeError
    977     # This aliases json.JSONDecodeError and simplejson.JSONDecodeError
--> 978     raise RequestsJSONDecodeError(e.msg, e.doc, e.pos)

JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

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.

Function calling in a chat session

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

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

[24]:
```
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",
    "properties": {
        "product": {
            "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,
                          ],
)
```

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.

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.

[26]:
```
model = GenerativeModel("gemini-2.0-flash-001",
                        generation_config={"temperature": 0},
                        tools=[retail_tool])
chat = model.start_chat()
```

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

[27]:
```
prompt = """
Do you have the Pixel 8 Pro in stock?
"""

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

[27]:
```
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.

In reality, you would execute function calls against an external system or database. 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.

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

[28]:
```
api_response = {"sku": "GA04834-US", "in_stock": "yes"}
```

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

[29]:
```
response = chat.send_message(
    Part.from_function_response(
        name="get_product_sku",
        response={
            "content": api_response,
        }
    ),
)
response.candidates[0].content.parts[0]
```

[29]:
```
text: "Yes, we have the Pixel 8 Pro in stock. The SKU is GA04834-US.\n"
```

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

[30]:
```
prompt = """
Where can I buy one near Mountain View, CA?
"""

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

[30]:
```
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.

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

[31]:
```
api_response = {"store": "1600 Amphitheatre Pkwy, Mountain View, CA 94043, US"}
```

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

[32]:
```
response = chat.send_message(
    Part.from_function_response(
        name="get_store_location",
        response={
            "content":  api_response,
        }
    ),
)
response.candidates[0].content.parts[0]
```

[32]:
```
text: "The closest store is located at 1600 Amphitheatre Pkwy, Mountain View, CA 94043, US.\n"
```

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

[33]:
```
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]
```

[33]:
```
function_call {
  name: "place_order"
  args {
    fields {
      key: "account"
      value {
        number_value: 101.0
      }
    }
    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:

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

[34]:
```
api_response = {"payment_status": "paid", "order_number": 12345, "est_arrival": "2 days"}
```

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

[35]:
```
response = chat.send_message(
    Part.from_function_response(
        name="place_order",
        response={
            "content":  api_response,
        }
    ),
)
response.candidates[0].content.parts[0]
```

[35]:
```
text: "OK. I have placed an order for a Pixel 8 Pro to be shipped to 1155 Borregas Ave, Sunnyvale, CA 94089, using account number 101. The order number is
12345, the payment status is paid, and the estimated arrival is in 2 days.\n"
```

[ ]: