Programming Generative AI Applications

In this module, you learn to ...

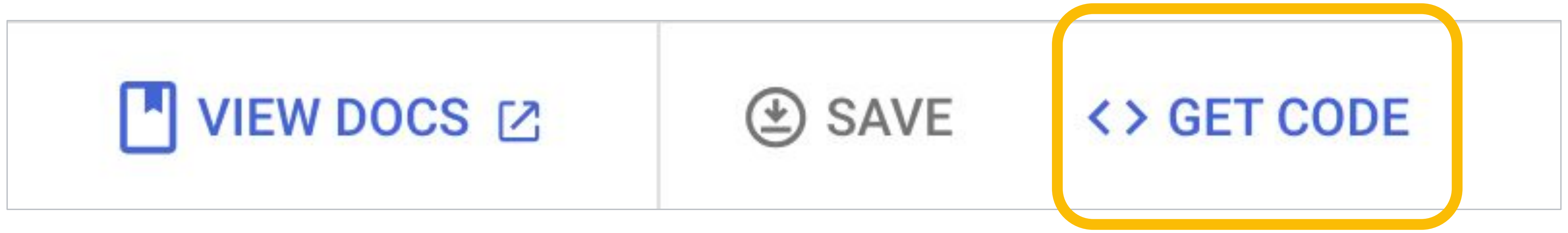
01 Program with the Gemini REST API

02 Program Jupyter Notebooks that use Gemini

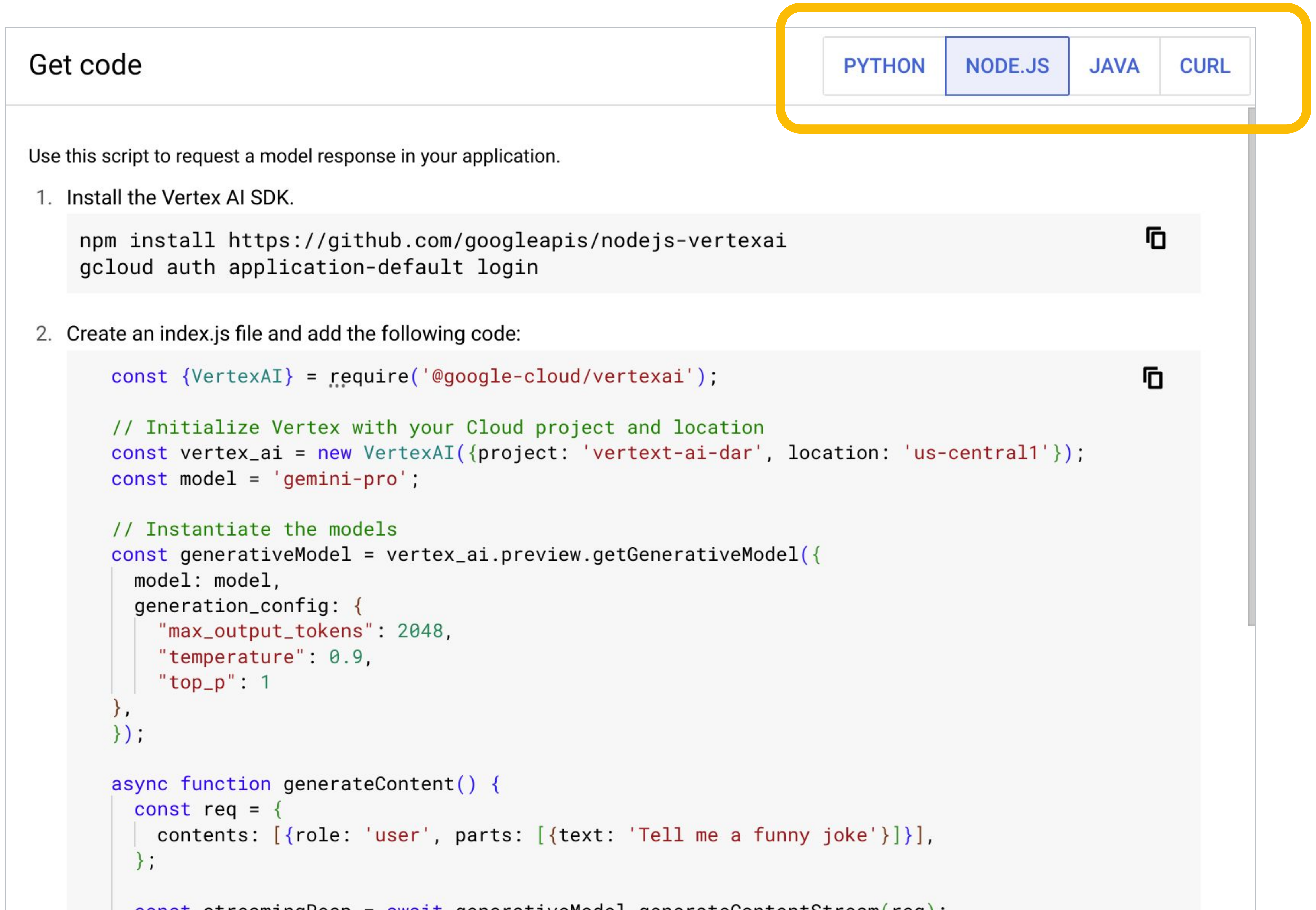
03 Add generative AI capabilities to your Python applications

Topics

For an easy start, use Get Code from Vertex AI Studio to provide template code



Get package installation and code with any parameters or safety settings you've adjusted in a few languages



Get code

PYTHON | NODE.JS | JAVA | CURL

Use this script to request a model response in your application.

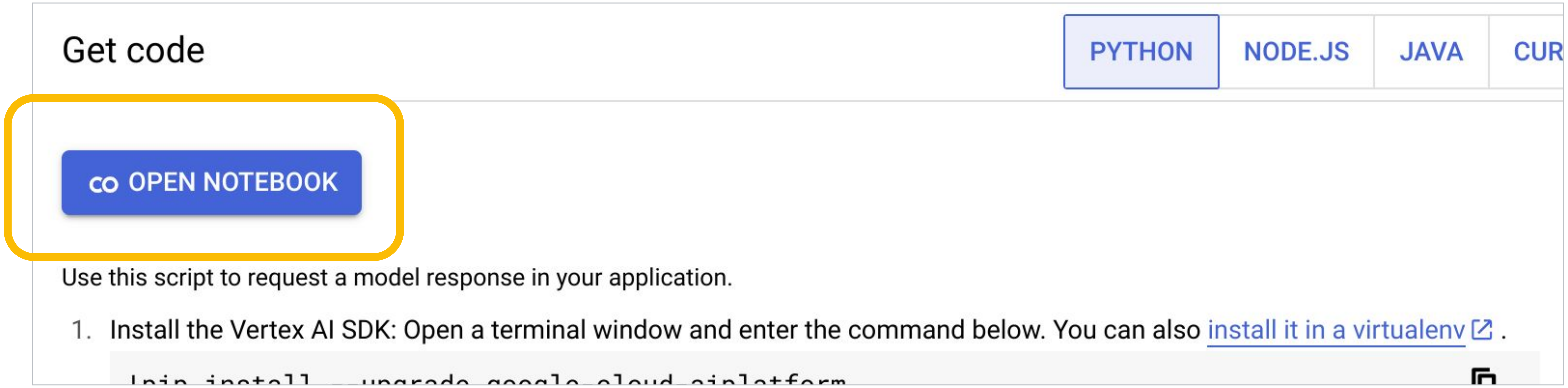
1. Install the Vertex AI SDK.

```
npm install https://github.com/googleapis/nodejs-vertexai
gcloud auth application-default login
```

2. Create an index.js file and add the following code:

```
const {VertexAI} = require('@google-cloud/vertexai');

// Initialize Vertex with your Cloud project and location
const vertex_ai = new VertexAI({project: 'vertext-ai-dar', location: 'us-central1'});
const model = 'gemini-pro';

// Instantiate the models
const generativeModel = vertex_ai.preview.getGenerativeModel({
  model: model,
  generation_config: {
    "max_output_tokens": 2048,
    "temperature": 0.9,
    "top_p": 1
},
});

async function generateContent() {
  const req = {
    contents: [{role: 'user', parts: [{text: 'Tell me a funny joke'}]}],
  };

  const streamingResp = await generativeModel.generateContentStream(req);
```

For Python code, the Open Notebook button will create a Colab Enterprise notebook for you



Use this script to request a model response in your application.

1. Install the Vertex AI SDK: Open a terminal window and enter the command below. You can also install it in a virtualenv .

```
!pip install --upgrade google-cloud-aiplatform
```

The basic pattern in Python: Installation

Use pip to install Google Cloud AI Platform. (Or add to your `requirements.txt` file)

```
pip install –quiet --upgrade google-cloud-aiplatform
```

The basic pattern in Python: Imports

Even though you installed using **`google-cloud-aiplatform`**, you will import **`vertexai`**

```
import vertexai
from vertexai.generative_models import (GenerativeModel,
                                        GenerationConfig,
                                        Image,
                                        Part,
                                        FinishReason)
```

The basic pattern in Python: Initialize Vertex AI

Initiliaze with a **`project`** and **`location`**.

```
vertexai.init(project=”my-project-id”, location=”us=central1”)
```

The basic pattern in Python: Optionally set Safety Settings

```python
safety_settings = [
    SafetySetting(
        category=SafetySetting.HarmCategory.HARM_CATEGORY_HATE_SPEECH,
        threshold=SafetySetting.HarmBlockThreshold.BLOCK_MEDIUM_AND_ABOVE
    ),
    SafetySetting(
        category=SafetySetting.HarmCategory.HARM_CATEGORY_DANGEROUS_CONTENT,
        threshold=SafetySetting.HarmBlockThreshold.BLOCK_LOW_AND_ABOVE
    ),
    SafetySetting(
        category=SafetySetting.HarmCategory.HARM_CATEGORY_SEXUALLY_EXPLICIT,
        threshold=SafetySetting.HarmBlockThreshold.BLOCK_MEDIUM_AND_ABOVE
    ),
    SafetySetting(
        category=SafetySetting.HarmCategory.HARM_CATEGORY_HARASSMENT,
        threshold=SafetySetting.HarmBlockThreshold.BLOCK_MEDIUM_AND_ABOVE
    ),
]
```

The basic pattern in Python: Instantiate a model

Instantiate the model. You can apply configuration at the model level or per query.

```
gen_config = GenerationConfig(
      temperature=temperature,
      top_p=top_p,
  )

model = GenerativeModel(“gemini-1.5-pro-001”,
                        generation_config = gen_config,
                        safety_settings=safety_settings)
```

The basic pattern in Python: Generation

```
response = model.generate_content(
      “My prompt or a list of prompt Parts.”,
      stream=False,
  )

print(response.text)
```

Use `generate_content()` function to call the model

Get the output using the text property of the response

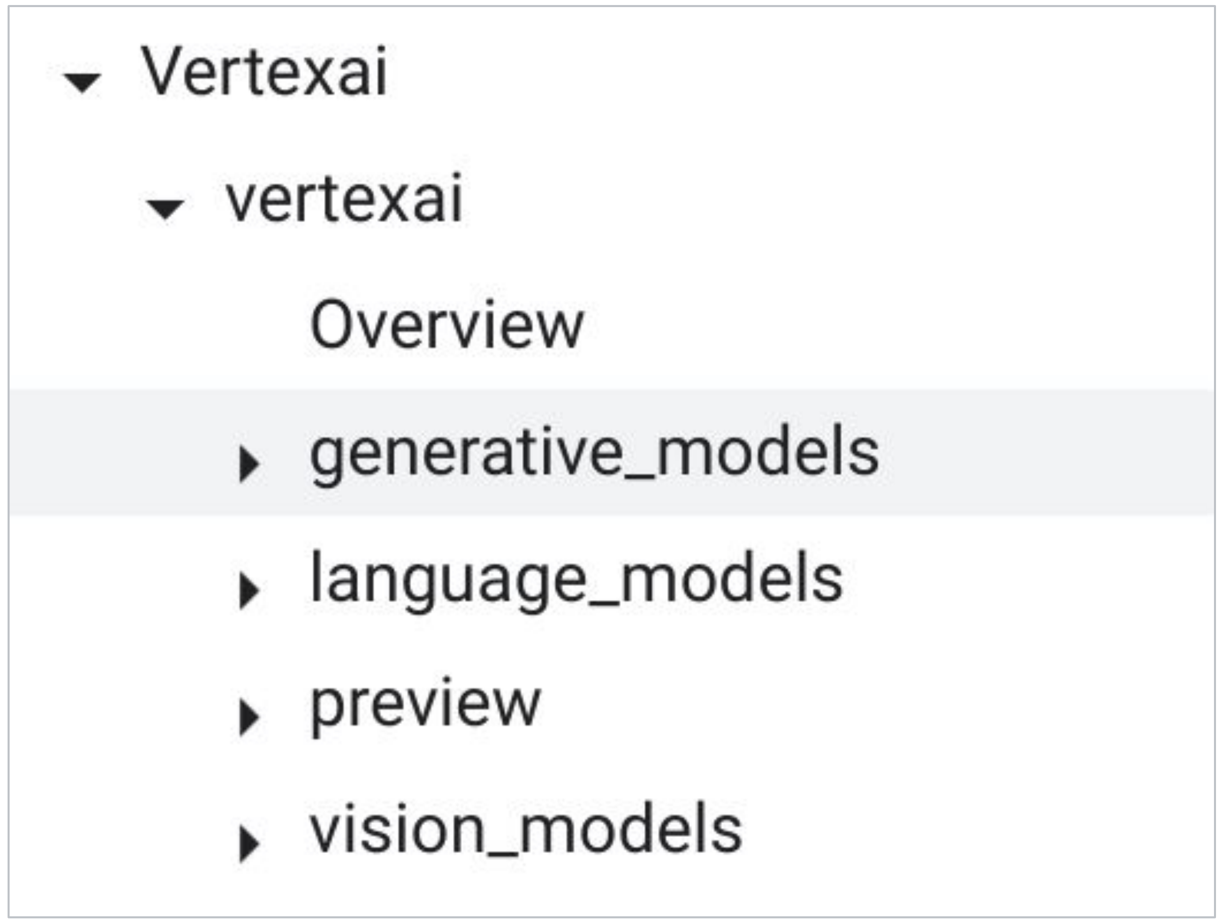
Using Gemini Pro Vision

```
from vertexai.generative_models import GenerativeModel, Image

multimodal_model = GenerativeModel("gemini-pro-vision")

image = Image.load_from_file("image.jpg")
prompt = "Describe this image?"

contents = [image, prompt]
responses = multimodal_model.generate_content(contents, stream=True)

for response in responses:
    print(response.text, end="")
```

The prompt along with the image(s) and/or video(s) are passed to the model

Python API Documentation

- See the documentation for API details
- The generative models section covers Gemini
- The language models section covers PaLM
- See vision models for Imagen



https://cloud.google.com/python/docs/reference/aiplatform/latest/vertexai

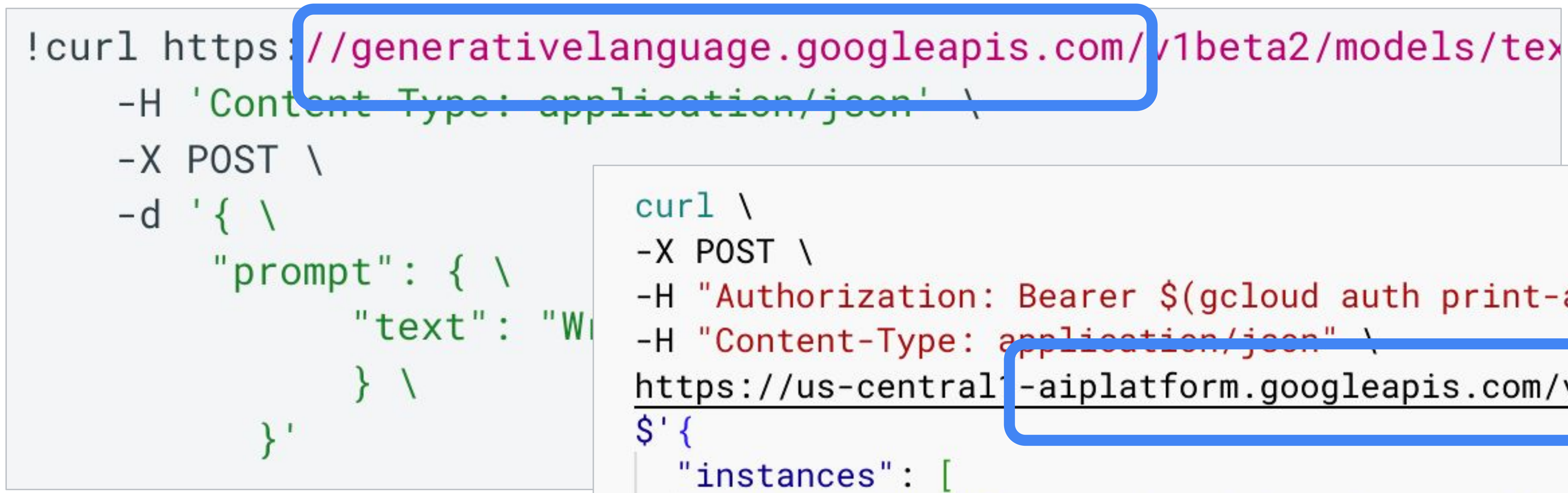
Topics

CURL code example (REST API)

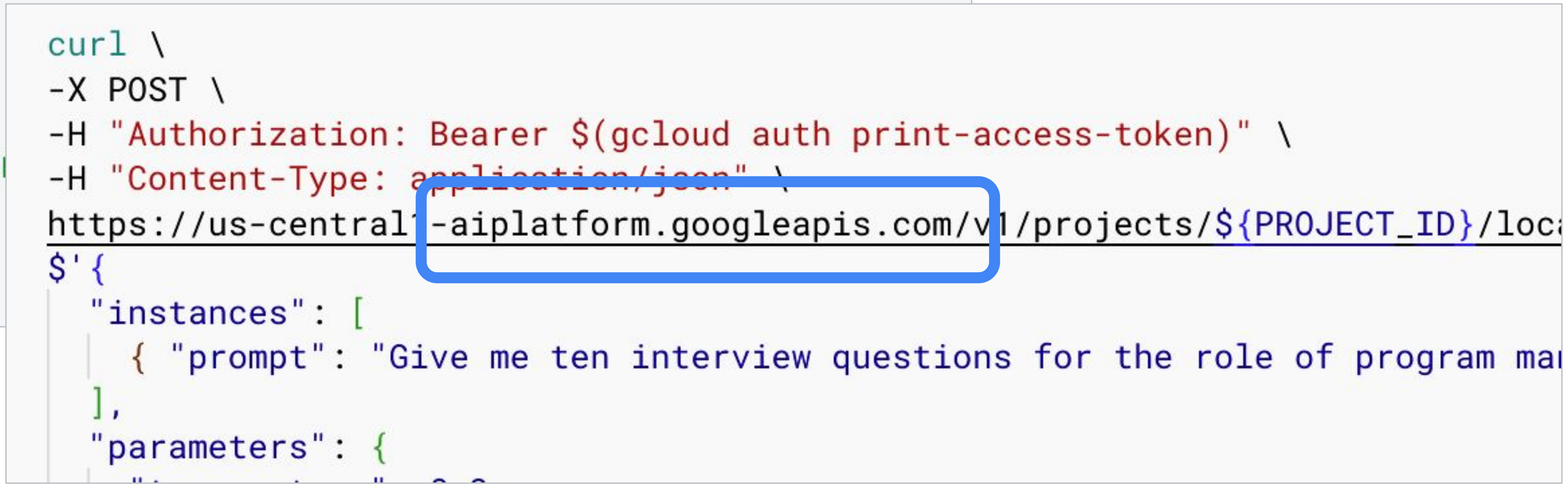
```
curl -X POST \
  -H "Authorization: Bearer $(gcloud auth print-access-token)" \
  -H "Content-Type: application/json" \

https://${LOCATION}-aiplatform.googleapis.com/v1/projects/${PROJECT_ID}/locations/\
${LOCATION}/publishers/google/models/${MODEL_ID}:generateContent \
-d '{
  "contents": [{
    ...
  }],
  "generationConfig": {
    ...
  },
  ...
}'
```

Be careful to use the Vertex AI API

- Google makes available two APIs for developing with GenAI apps using Gemini
  - One API is made available to the general public
  - For Enterprise applications, make sure to use the Google Cloud Vertex AI API
- Examine the endpoints
  - The public API uses **`generativelanguage.googleapis.com`**
  - The enterprise API uses **`aiplatform.googleapis.com`**



```
!curl https://generativelanguage.googleapis.com/v1beta2/models/tex
    -H 'Content-Type: application/json' \
    -X POST \
    -d '{ \
        "prompt": { \
            "text": "Wr
            } \
        }'
```



```
curl \
-X POST \
-H "Authorization: Bearer $(gcloud auth print-access-token)" \
-H "Content-Type: application/json" \
https://us-central1-aiplatform.googleapis.com/v1/projects/${PROJECT_ID}/loca
$'{
  "instances": [
    { "prompt": "Give me ten interview questions for the role of program ma
  ],
  "parameters": {
```

An authorization token identifies the caller of the API

- Created using the Google Cloud CLI or the Cloud Console
  - The gcloud CLI must be initialized with either a user or service account
- Set the Authorization header variable with the token generated using gcloud

```
curl \
-X POST \
-H "Authorization: Bearer $(gcloud auth print-access-token)" \
-H "Content-Type: application/json" \
"https://${API_ENDPOINT}/v1/projects/${PROJECT_ID}/locations/${LOCATION_ID}/publis
hers/google/models/${MODEL_ID}:streamGenerateContent" -d '@request.json'
<<some code omitted>>
```

REST API response (code omitted for space)

```
[{"candidates": [{
      "content": {
        "role": "model",
        "parts": [{
            "text": "What do you"
          }]},
      "safetyRatings": [
        {
          "category": "HARM_CATEGORY_HATE_SPEECH",
          "probability": "NEGLIGIBLE",
          "probabilityScore": 0.08977328,
          "severity": "HARM_SEVERITY_NEGLIGIBLE",
          "severityScore": 0.075995214}...
      "finishReason": "STOP",]}],
  "usageMetadata": {
    "promptTokenCount": 4,
    "candidatesTokenCount": 14,
    "totalTokenCount": 18}}]
```

The response is streamed in parts

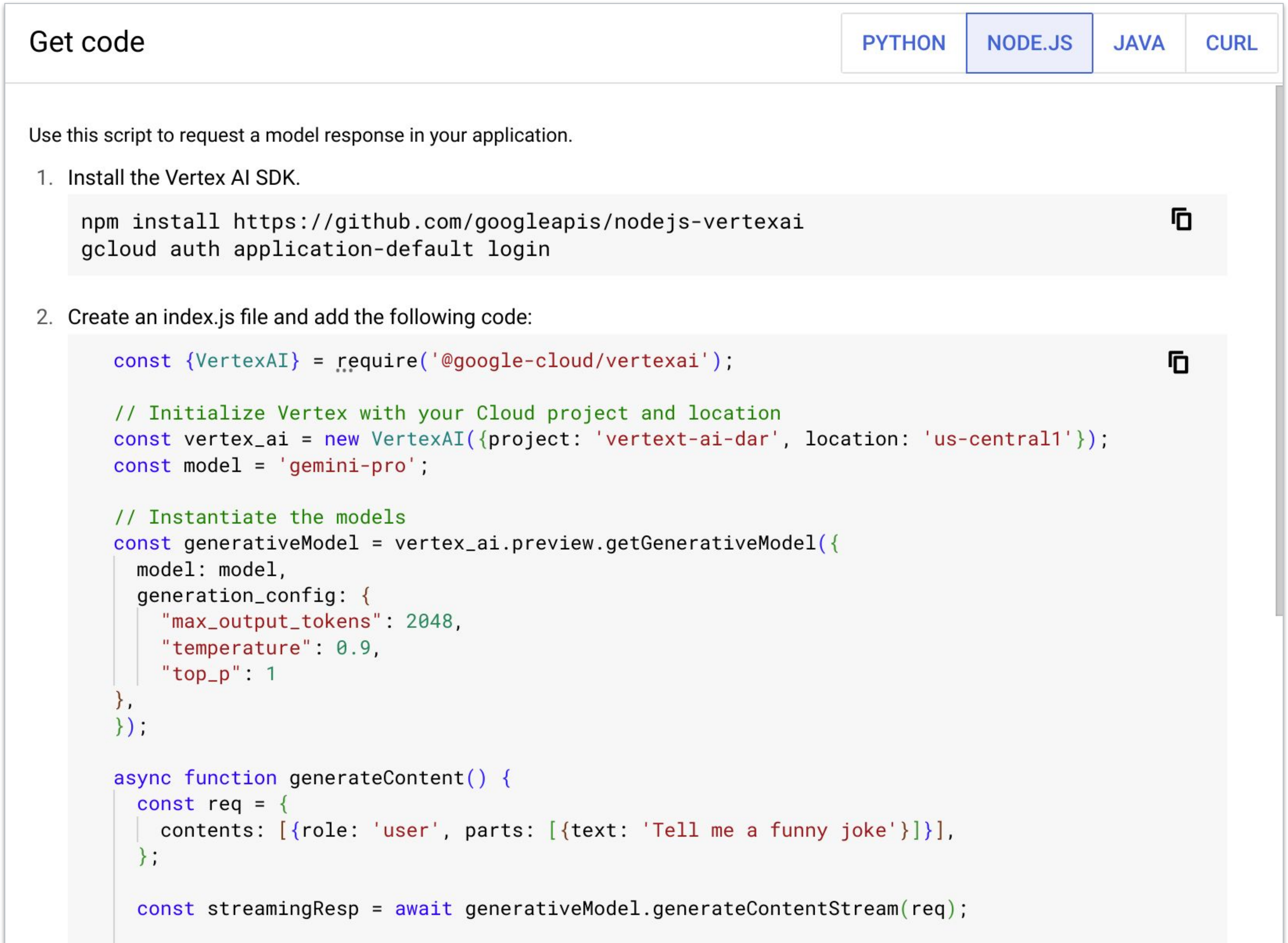
Safety Ratings can be used to detect inappropriate content

Input and output tokens determine the cost

Safety categories score probability and severity. By default, safety settings block on probability.

```
"safetyRatings": [
        {
          "category": "HARM_CATEGORY_HATE_SPEECH",
          "probability": "NEGLIGIBLE",
          "probabilityScore": 0.344223,
          "severity": "HARM_SEVERITY_LOW",
          "severityScore": 0.23510839
        },
        {
          "category": "HARM_CATEGORY_DANGEROUS_CONTENT",
          "probability": "NEGLIGIBLE",
          "probabilityScore": 0.20753574,
          "severity": "HARM_SEVERITY_LOW",
          "severityScore": 0.2562732
        },...
```

JavaScript code example



Get code

PYTHON NODE.JS JAVA CURL

Use this script to request a model response in your application.

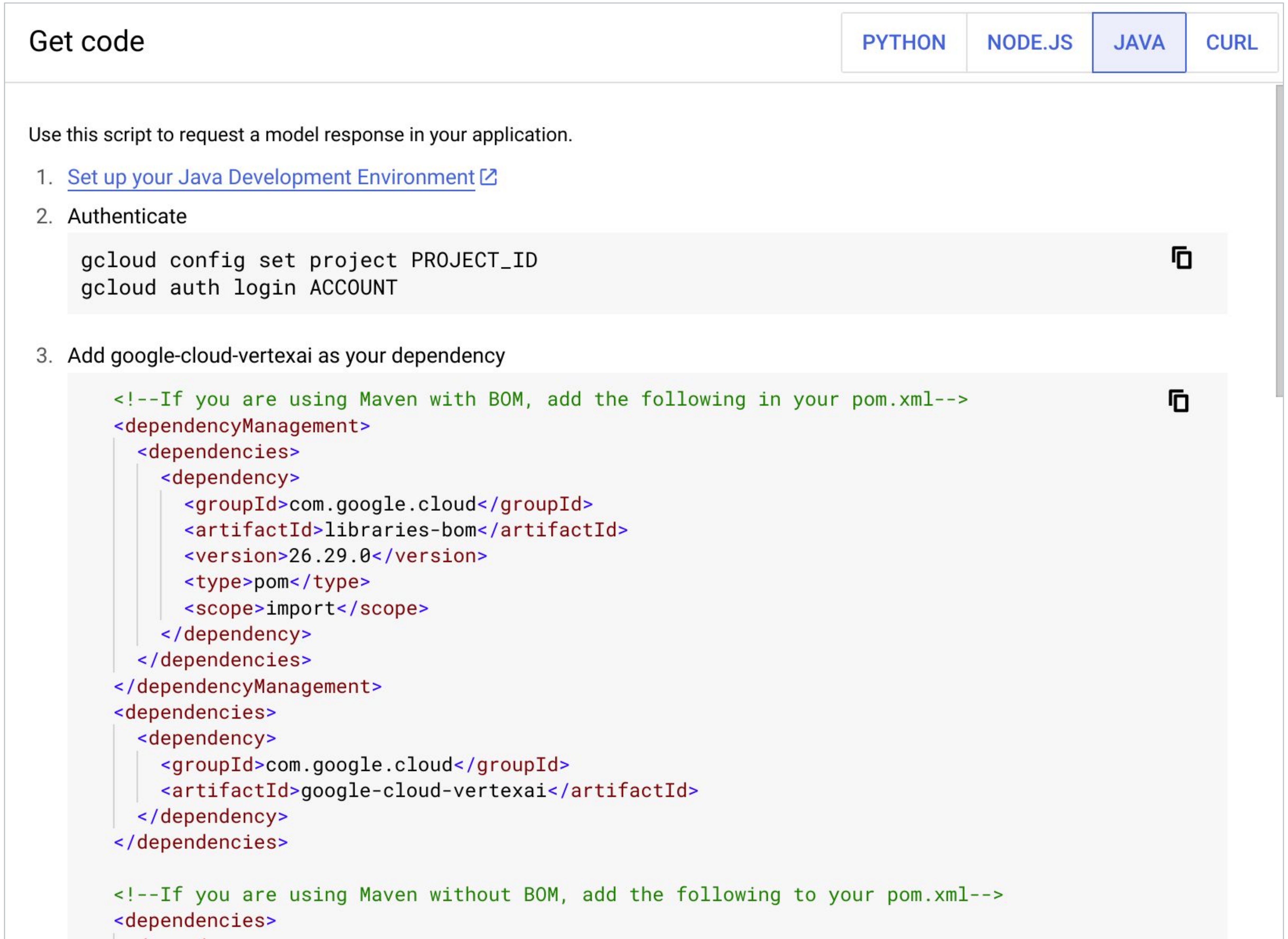
1. Install the Vertex AI SDK.

```
npm install https://github.com/googleapis/nodejs-vertexai
gcloud auth application-default login
```

2. Create an index.js file and add the following code:

```
const {VertexAI} = require('@google-cloud/vertexai');

// Initialize Vertex with your Cloud project and location
const vertex_ai = new VertexAI({project: 'vertext-ai-dar', location: 'us-central1'});
const model = 'gemini-pro';

// Instantiate the models
const generativeModel = vertex_ai.preview.getGenerativeModel({
  model: model,
  generation_config: {
    "max_output_tokens": 2048,
    "temperature": 0.9,
    "top_p": 1
},
});

async function generateContent() {
  const req = {
    contents: [{role: 'user', parts: [{text: 'Tell me a funny joke'}]}],
  };

  const streamingResp = await generativeModel.generateContentStream(req);
```

Java code example



Use this script to request a model response in your application.

1. Set up your Java Development Environment
2. Authenticate

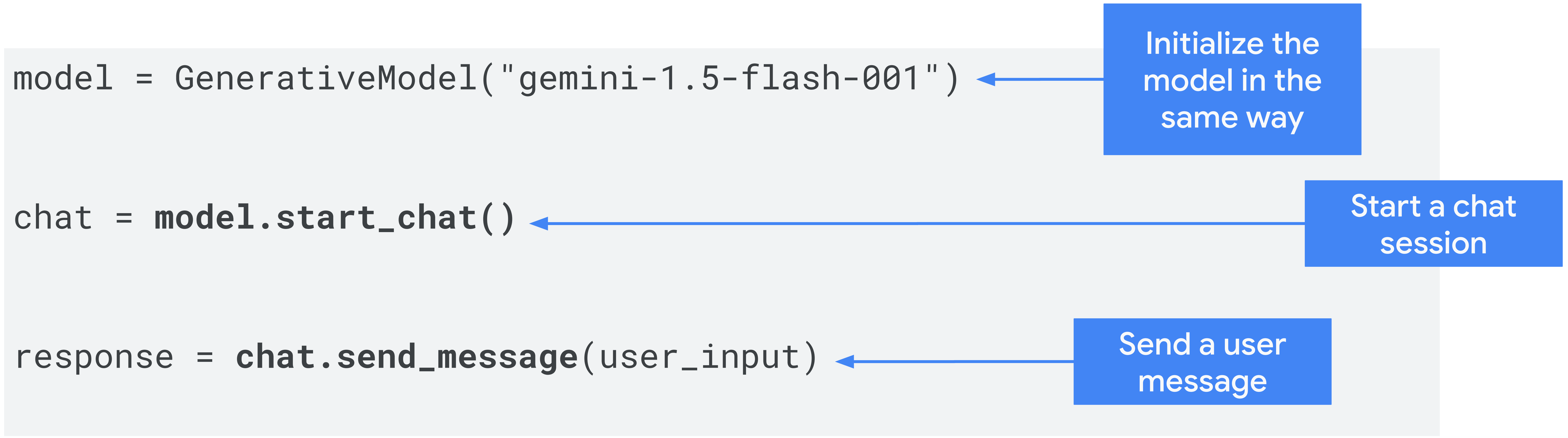
```
gcloud config set project PROJECT_ID
gcloud auth login ACCOUNT
```

3. Add google-cloud-vertexai as your dependency

```
<!--If you are using Maven with BOM, add the following in your pom.xml-->
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.google.cloud</groupId>
      <artifactId>libraries-bom</artifactId>
      <version>26.29.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>com.google.cloud</groupId>
    <artifactId>google-cloud-vertexai</artifactId>
  </dependency>
</dependencies>

<!--If you are using Maven without BOM, add the following to your pom.xml-->
<dependencies>
```

Topics

Initializing a Chat session



```
model = GenerativeModel("gemini-1.5-flash-001")

chat = model.start_chat()

response = chat.send_message(user_input)
```

Managing User Sessions

- In a chat, the history of the conversation needs to be maintained per user
  - The `ChatSession` object has a `message_history` property
- Create a session variable with the history for each user
  - Reinitialize the chat with every request setting the message history property
- In Python Flask, sessions are stored in the client browser, so this is a scalable solution

```
response = chat.send_message(input)
session["chat_history"] = chat.message_history
```

Need to convert the history in a session variable into ChatMessage objects

```
if 'chat_history' in session:
    chat_history = [ChatMessage(content=items["content"], author=items["author"])
                    for items in session["chat_history"]]
    parameters["message_history"] = chat_history
```

Streaming

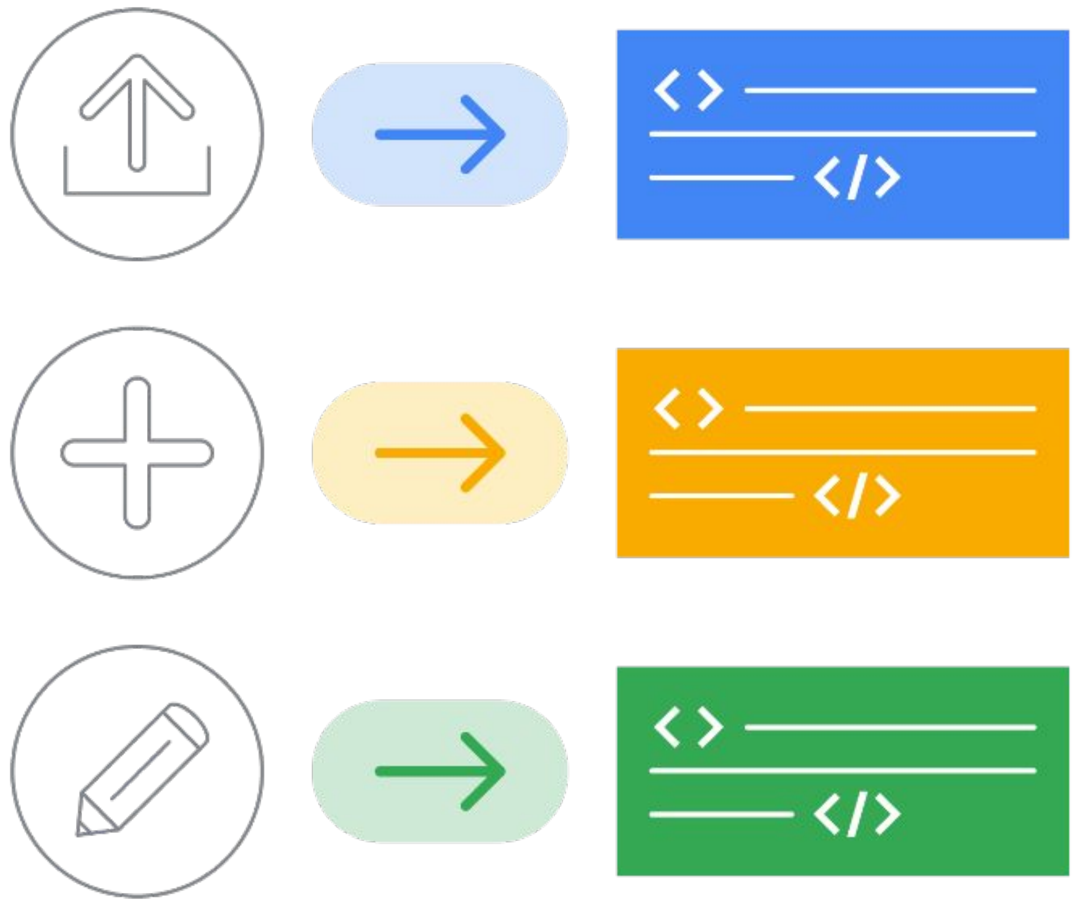
```
gemini_model = GenerativeModel(MODEL_ID)
model_response = gemini_model.generate_content(“Tell me a fable about friendship”,
                                               stream=True)

for response in responses:
    print(response.text)
```

Topics

Gemini implements Function Calling

Steps:

1. Add one or more functions to your application
2. Create a Function Declaration for each function
3. Add the Function Declarations to a Tool
4. Add one or more Tools to the Model
5. The model will decide what function to call and return the function name and parameters in its response
6. Call the function and return the results back to the model



Step 1: Add one or more functions to your application

- Add external functions to do whatever your agent might need to do to complete its task
    - Get external data
    - Call an API
    - Make a web request
    - etc.

```
def multiply(a:float, b:float):
    """returns a * b."""
    print("Calling Multiply function")
    return a * b
```

Step 2: Create a Function Declaration for each function

```
multiply_info = FunctionDeclaration(
    name="multiply",
    description="Multiplies two numbers and returns the result",
    parameters={
        "type": "object",
        "properties": {
            "a": {"type": "number", "description": "First number"},
            "b": {"type": "number", "description": "Second number"}
        },
    },
)
```

The Function Declaration describes the function and its parameters

Step 3: Add the Function Declarations to a Tool

```
math_tool = Tool(
    function_declarations=[
        multiply_info,
        add_info
    ],
)
```

Tools contain 1 or more Function Declarations

Step 4: Add one or more Tools to the Model

```
model = GenerativeModel(
    "gemini-1.5-pro-001",
    system_instruction=["""Answer the user's question,
    but do not do any math yourself."""],
    tools=[math_tool]
  )
```

The system instruction tells the model to use the functions

Specify the tools when creating the model

Step 5: The model will decide what function to call and return the function name and parameters in its response

```
response = chat.send_message("I have 7 pizzas each with 16 slices. How many
slices do I have?")

print(response)
```

```
candidates {
   content {
      role: “model”
      parts {
         function call {
            name: “multiply”
            args {
               fields {
                  key: “a”
                  value {
                     number_value: 7.0
                  }
               }
               fields {
                  key: “b”
                  value {
                     number_value: 16.0
                  }
```

Need to call the multiply function to answer the question

The model returns the function the needs to be called and its arguments

Step 6: Call the function and return the results back to the model

```
def handle_response(response):

  if response.candidates[0].function_calls:
    function_call = response.candidates[0].function_calls[0]
  else:
    print(response.text)
    return

  if function_call.name == "multiply":
      a = function_call.args["a"]
      b = function_call.args["b"]

      # Call your function
      result = multiply(a, b)

      response = chat.send_message("{0}".format(result))
```

Check if there is a function that needs to be called

Extract the arguments

Call the function

Send the results back to the model

Best practices with function calling

Use a manageable number of functions: We've found that 3-5 distinct function declarations gives the generative model a sensible range of functions to consider at runtime without causing too much non-determinism due to a larger number of possibilities.

Improve accuracy when selecting functions: Write clear and verbose function descriptions to help the model better understand the intent of the function to match user queries.

Improve accuracy in entity and parameter extraction: Write clear and verbose parameter descriptions to help the model better predict the parameter value.

Best practices with function calling (cont.)

Specify types as much as possible: Use strongly typed parameters from the OpenAPI schema when possible to reduce model inaccuracy.

Aim for determinism in function calling: Use `temperature=0` or a low value to instruct the model to generate more confident results and reduce hallucinations.

Provide few-shot examples: Provide few-shot examples to boost the performance by embedding examples in the function descriptions.

Topics

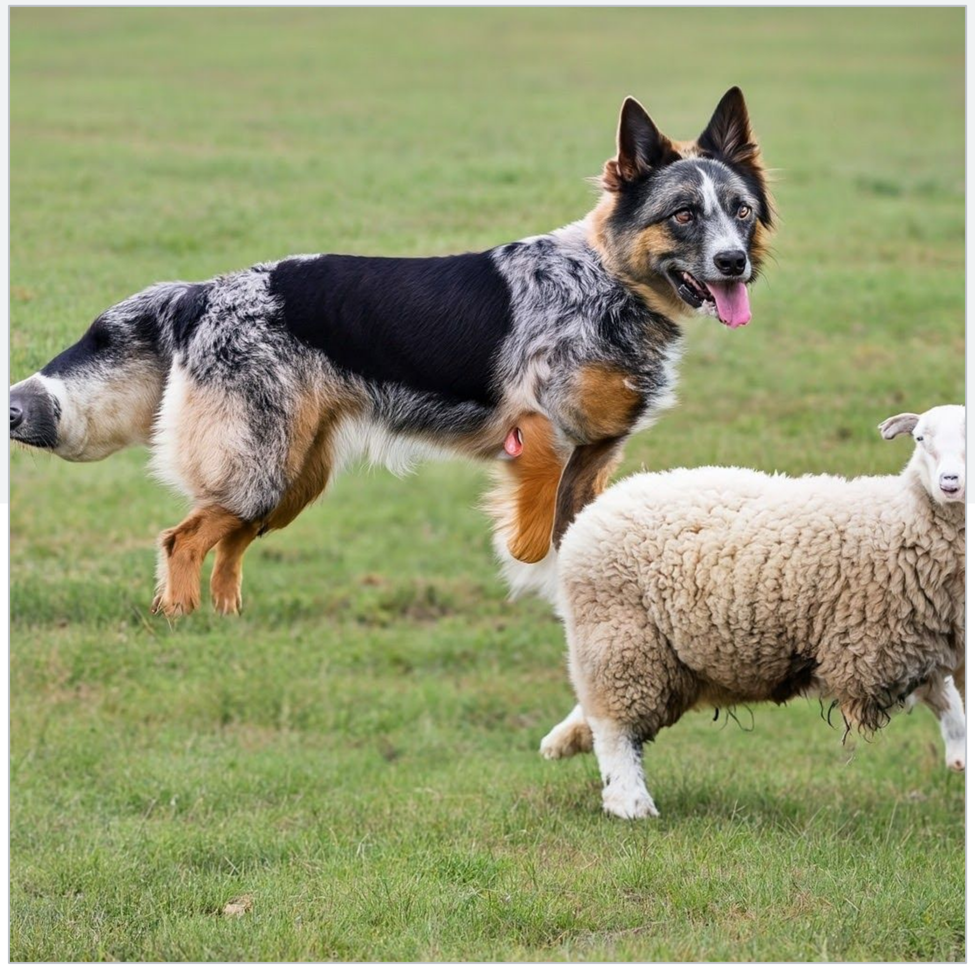
Using Imagen for image generation

```
from vertexai.vision_models import ImageGenerationModel, Image

model = ImageGenerationModel.from_pretrained("imagegeneration@002"
response = model.generate_images(
    prompt="Australian Shepherd herding sheep in a field, focus on the dog",
    # Optional:
    number_of_images=1
)
response[0].show()
response[0].save("shepherd.png")
```

Using Imagen for image captioning

```
from vertexai.vision_models import ImageCaptioningModel, Image

model = ImageCaptioningModel.from_pretrained("imagetext@001")
image = Image.load_from_file("shepherd.png")
captions = model.get_captions(
    image=image,
    number_of_results=3,
    language="en",
)
for caption in captions:
  print(caption)
```

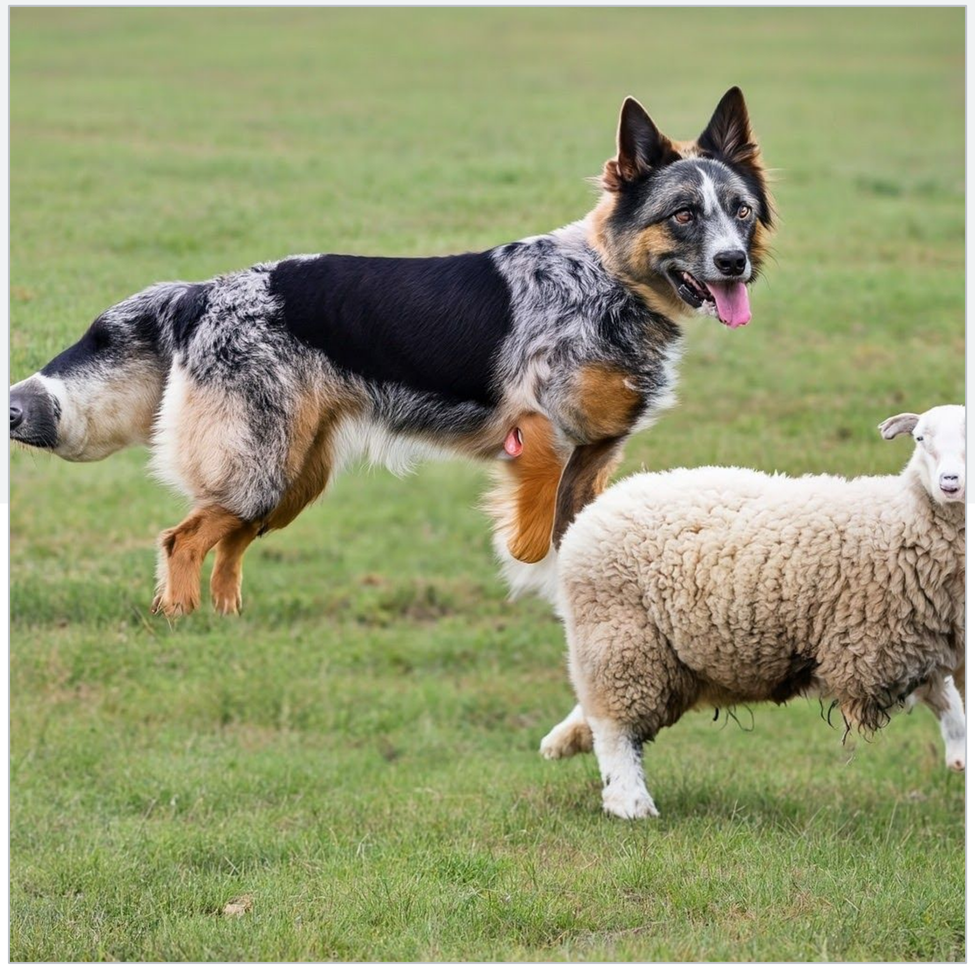
```
a dog is jumping over a sheep in a field
a dog jumping over a sheep in a field
a dog is jumping over a sheep in a grassy field
```



Using Imagen for image Q&A

```
from vertexai.vision_models import ImageQnAModel, Image

model = ImageQnAModel.from_pretrained("imagetext@001")
image = Image.load_from_file("shepherd.png")
answers = model.ask_question(
    image=image,
    question="what kind of dog is in this picture?",
    # Optional:
    number_of_results=3,
)
print(answers)
```

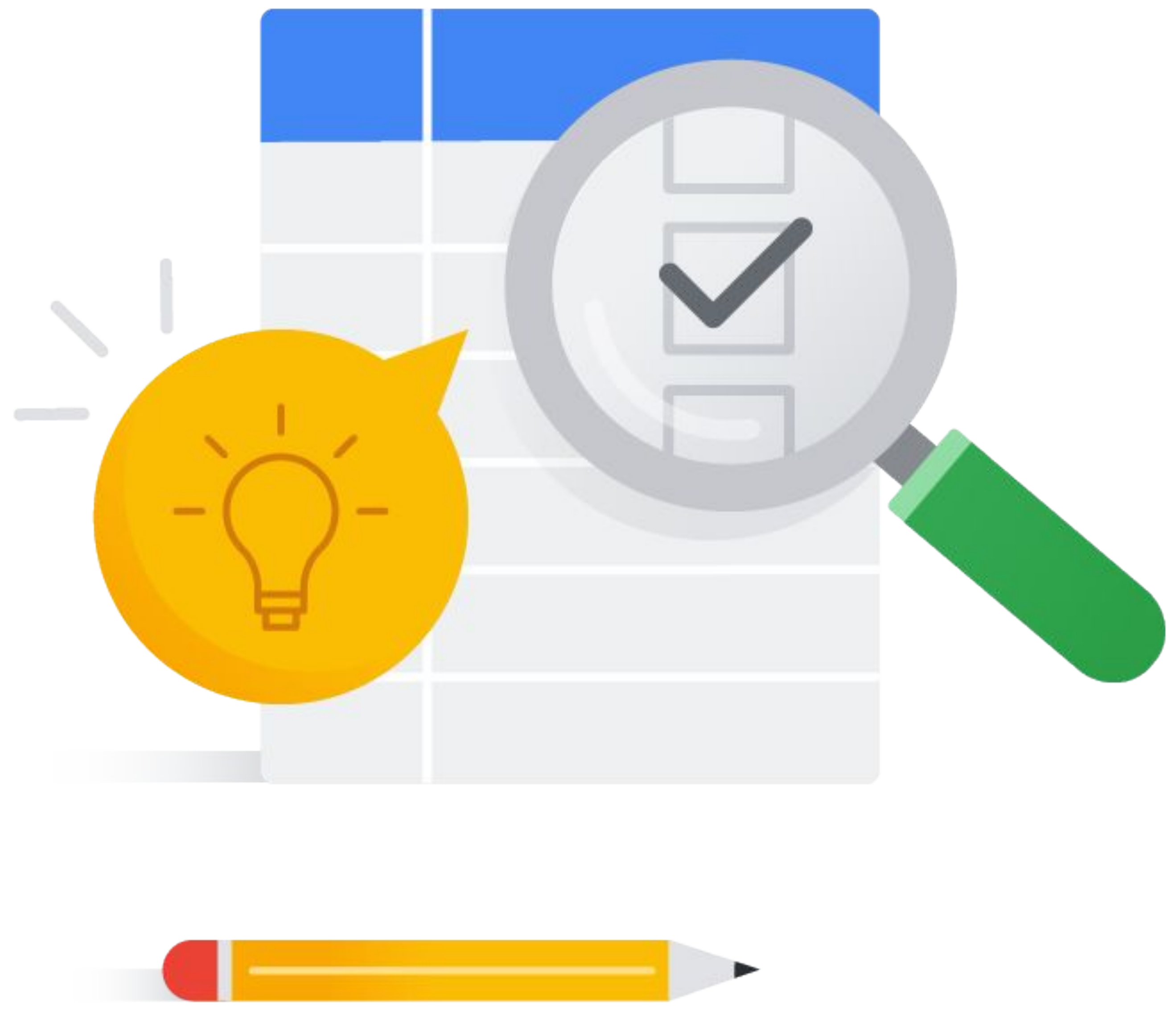
```
[‘border collie’, ‘shepherd’, ‘collie’]
```



Lab

1 hour

Lab: Introduction to Function Calling with Gemini



In this module, you learned to ...

01 Program with the Gemini REST API

02 Program Jupyter Notebooks that use Gemini

03 Add generative AI capabilities to your Python applications

Questions and answers

Quiz question

Which of the following methods can you use to authorize PaLM API requests from an application?

A: Obtain an authorization token and pass it in the header of the request

B: Assign a service account to your application runtime environment

C: Use a service account key

D: All of the above depending on the specific use case

Quiz question

Which of the following methods can you use to authorize PaLM API requests from an application?

A: Obtain an authorization token and pass it in the header of the request

B: Assign a service account to your application runtime environment

C: Use a service account key

D: All of the above depending on the specific use case

Quiz question

What is the main difference between a Text Generation and Chat program?

A: Text generation uses a large language model, chat does not

B: Chat uses a large language model, text generation does not

C: With text generation you have to maintain the history

D: With chat you have to maintain the history

Quiz question

What is the main difference between a Text Generation and Chat program?

A: Text generation uses a large language model, chat does not

B: Chat uses a large language model, text generation does not

C: With text generation you have to maintain the history

D: With chat you have to maintain the history

Google Cloud