03

Programming Gen AI Applications

In this module, you learn to ...

01 Program with the PaLM REST API

02 Program Jupyter Notebooks that use the PaLM API

03 Add GenAI capabilities to your Python applications

Topics

01 Getting Started

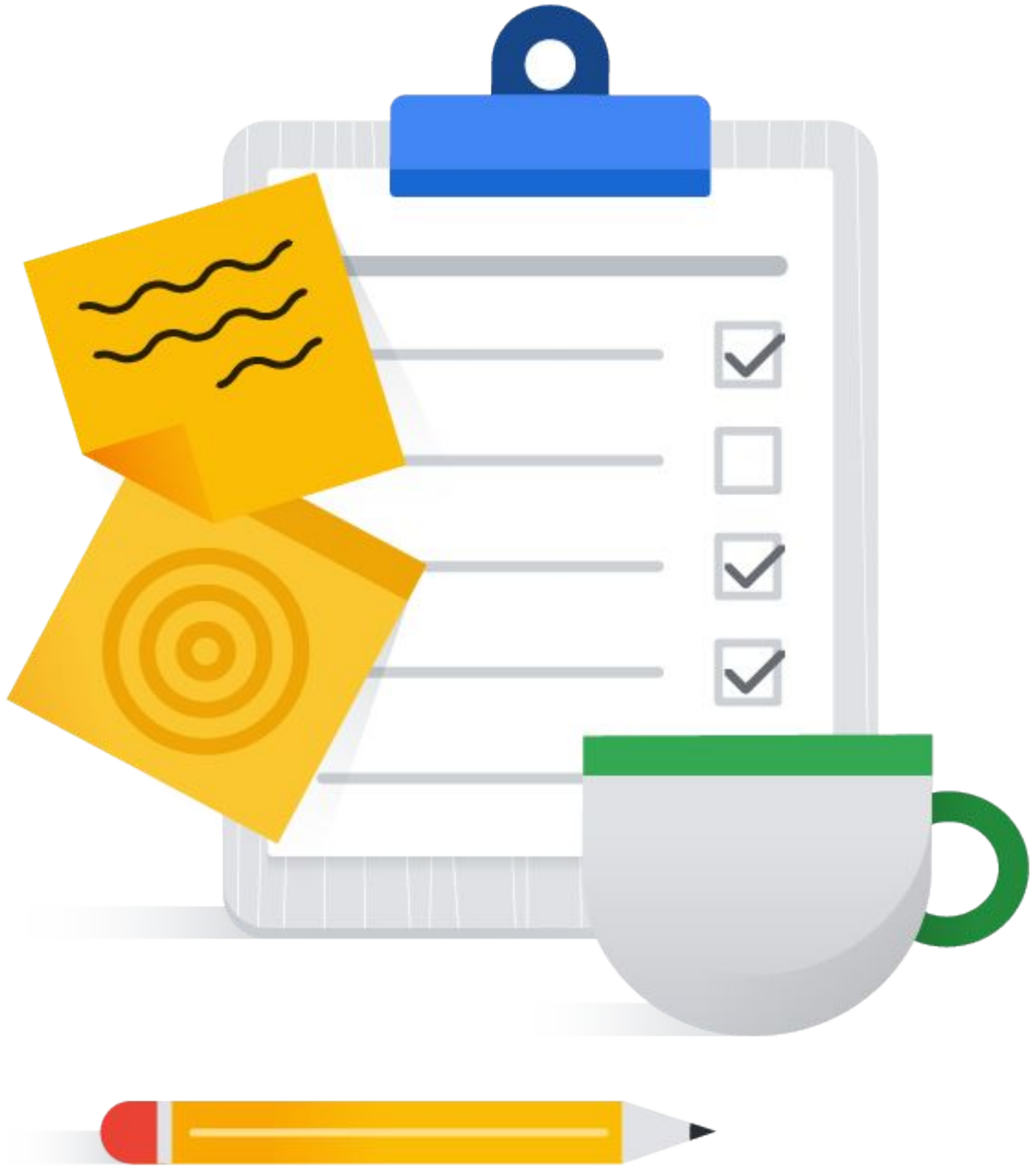
02 Python Language API

03 Programming Text Generation Applications

04 Programming Chat Applications

05 Large Document Processing

06 Imagen and Gemini



PaLM is a Large Language Model (LLM)

- LLMs are very sophisticated autocomplete applications
  - They learn patterns from large amounts of text
  - Use those patterns to generate text
- When generating text they calculate the next most likely tokens (words)
  - They aren’t smart; it’s math and statistics
- PaLM can generate text with two basic services
  - Text service for single request interactions
  - Chat service is for interactive, multi-turn interactions

To use the PaLM API, authenticate your application

- Obtain an authorization token
- Run the application using a service account

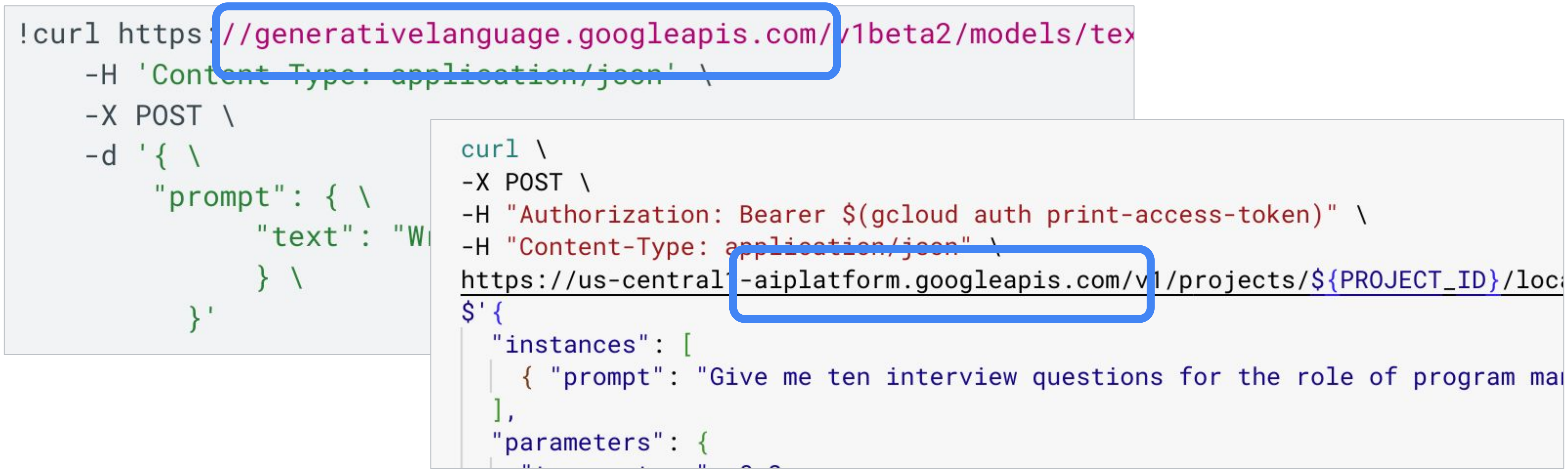
An authorization token identifies the caller of an API

- Created using the Google Cloud CLI
    - 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 $(gcloud auth print-access-token)" \
-H "Content-Type: application/json" \
"https://us-central1-aiplatform.googleapis.com/v1/projects/${PROJECT_ID}/locations/${LOCAT
ION_ID}/publishers/google/models/${MODEL_ID}:predict" -d \
$'{
    "instances": [
<<code omitted>>
```

Be careful when using the PaLM API

- Google makes available two APIs for developing with PaLM
  - 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`

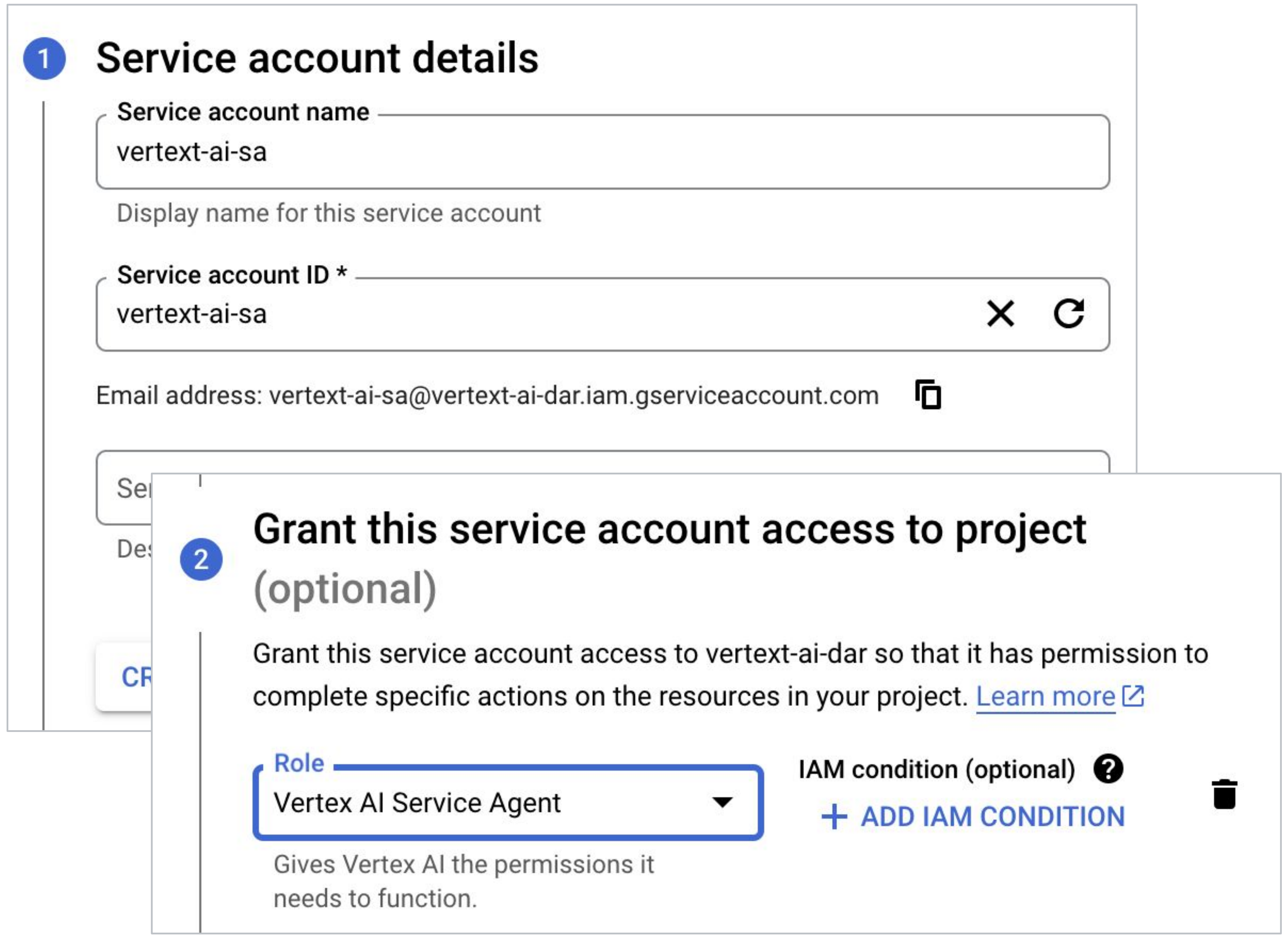


```
!curl https://generativelanguage.googleapis.com/v1beta2/models/tex
    -H 'Content-Type: application/json' \
    -X POST \
    -d '{ \
        "prompt": { \
              "text": "W
              } \
        }'
```

```
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}/loc
$'{
  "instances": [
    { "prompt": "Give me ten interview questions for the role of program ma
  ],
  "parameters": {
```

If running an application in Google Cloud, assign a service account to the runtime

- Create a service account using **IAM**
  - Assign the **Vertex AI Service Agent** role
  - Use the service account to identify the runtime
- If using Cloud Run, App Engine, or Cloud Functions, the runtime will use the Compute Engine Default Service Account by default
  - This will work as it uses the Editor role
  - However, it violates principle of least privilege
- You can also download Service Account keys to authenticate programs that use the language client libraries



1 Service account details

Service account name: vertext-ai-sa

Display name for this service account

Service account ID *: vertext-ai-sa

Email address: vertext-ai-sa@vertext-ai-dar.iam.gserviceaccount.com

2 Grant this service account access to project (optional)

Grant this service account access to vertext-ai-dar so that it has permission to complete specific actions on the resources in your project. Learn more

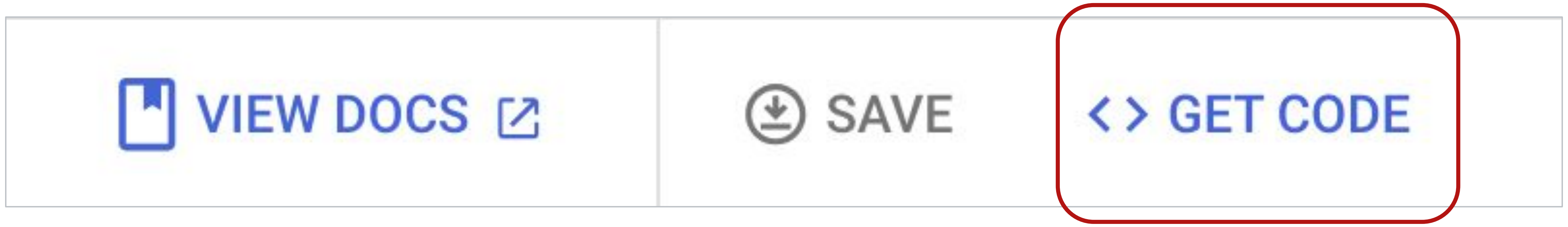
Role: Vertex AI Service Agent

Gives Vertex AI the permissions it needs to function.

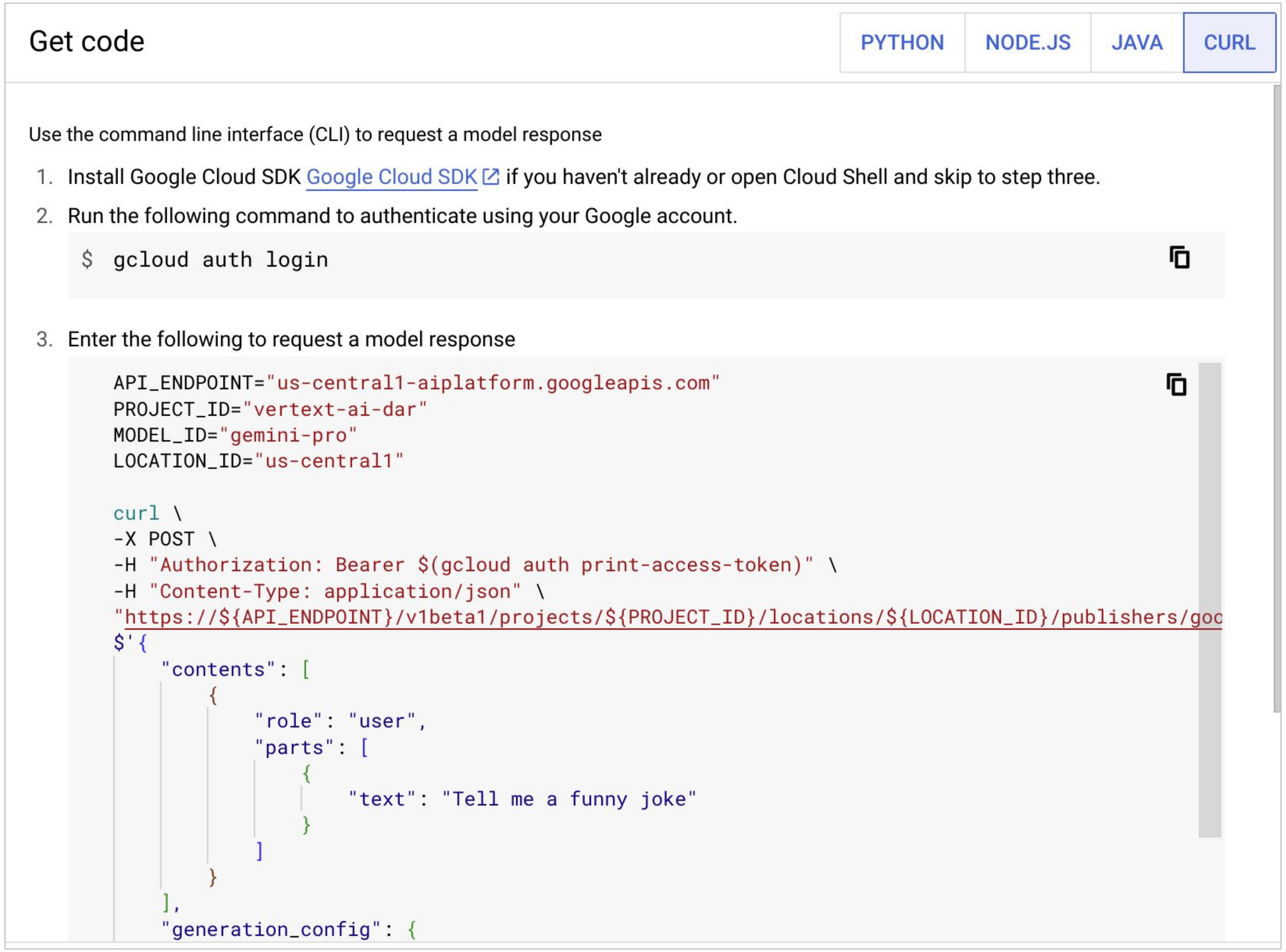
IAM condition (optional)

Generating code with Vertex AI Studio

- In Vertex AI Studio, click the **Get Code** button
  - Returns the code in Python, iPython, and cURL



CURL code example (REST API)



Use the command line interface (CLI) to request a model response

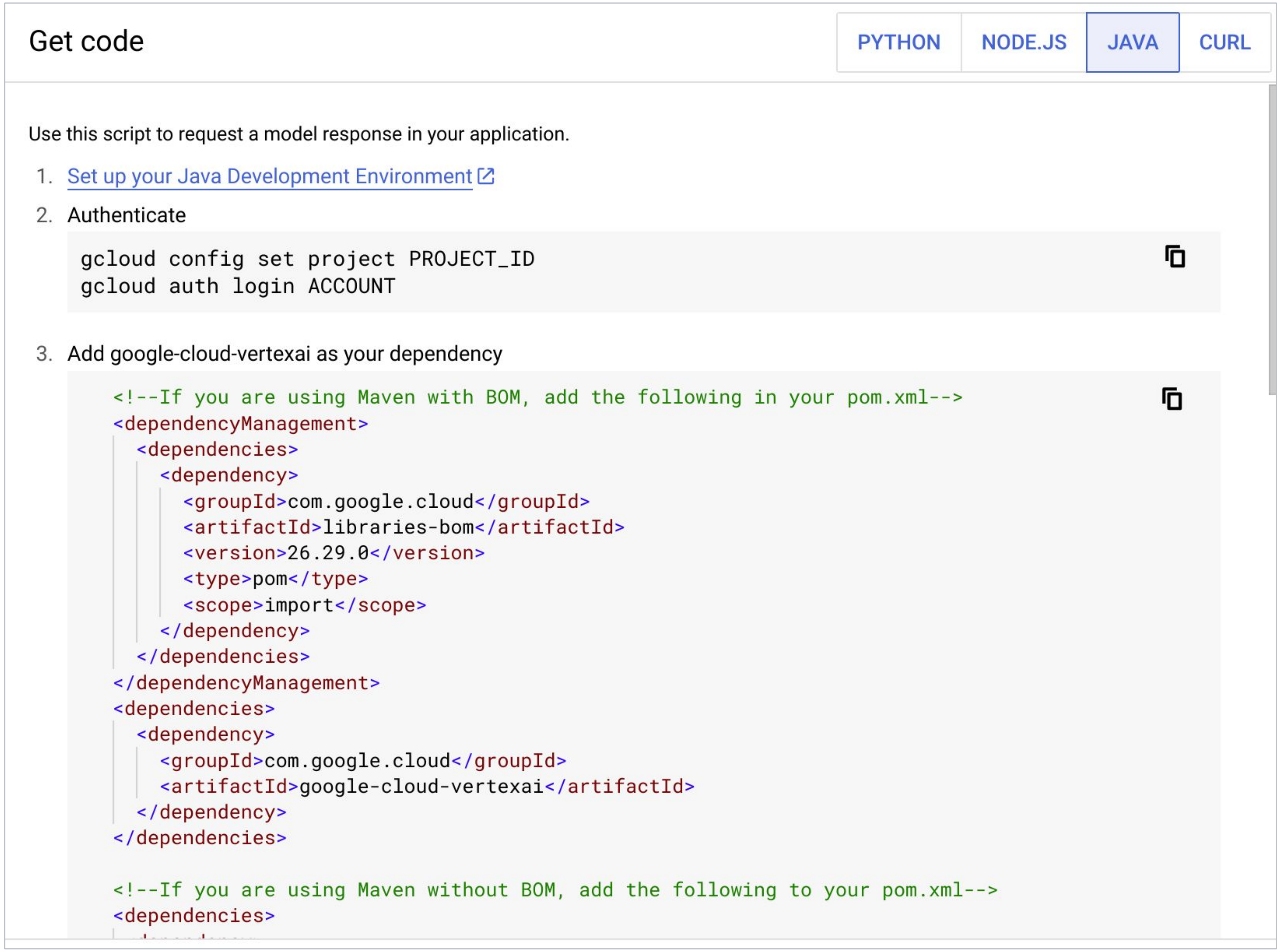
1. Install Google Cloud SDK Google Cloud SDK if you haven't already or open Cloud Shell and skip to step three.
2. Run the following command to authenticate using your Google account.

```
$ gcloud auth login
```

3. Enter the following to request a model response

```
API_ENDPOINT="us-central1-aiplatform.googleapis.com"
PROJECT_ID="vertext-ai-dar"
MODEL_ID="gemini-pro"
LOCATION_ID="us-central1"

curl \
-X POST \
-H "Authorization: Bearer $(gcloud auth print-access-token)" \
-H "Content-Type: application/json" \
"https://${API_ENDPOINT}/v1beta1/projects/${PROJECT_ID}/locations/${LOCATION_ID}/publishers/goc
$'{
    "contents": [
        {
            "role": "user",
            "parts": [
                {
                    "text": "Tell me a funny joke"
                }
            ]
        }
    ],
    "generation_config": {
```

Java code example



Get code

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

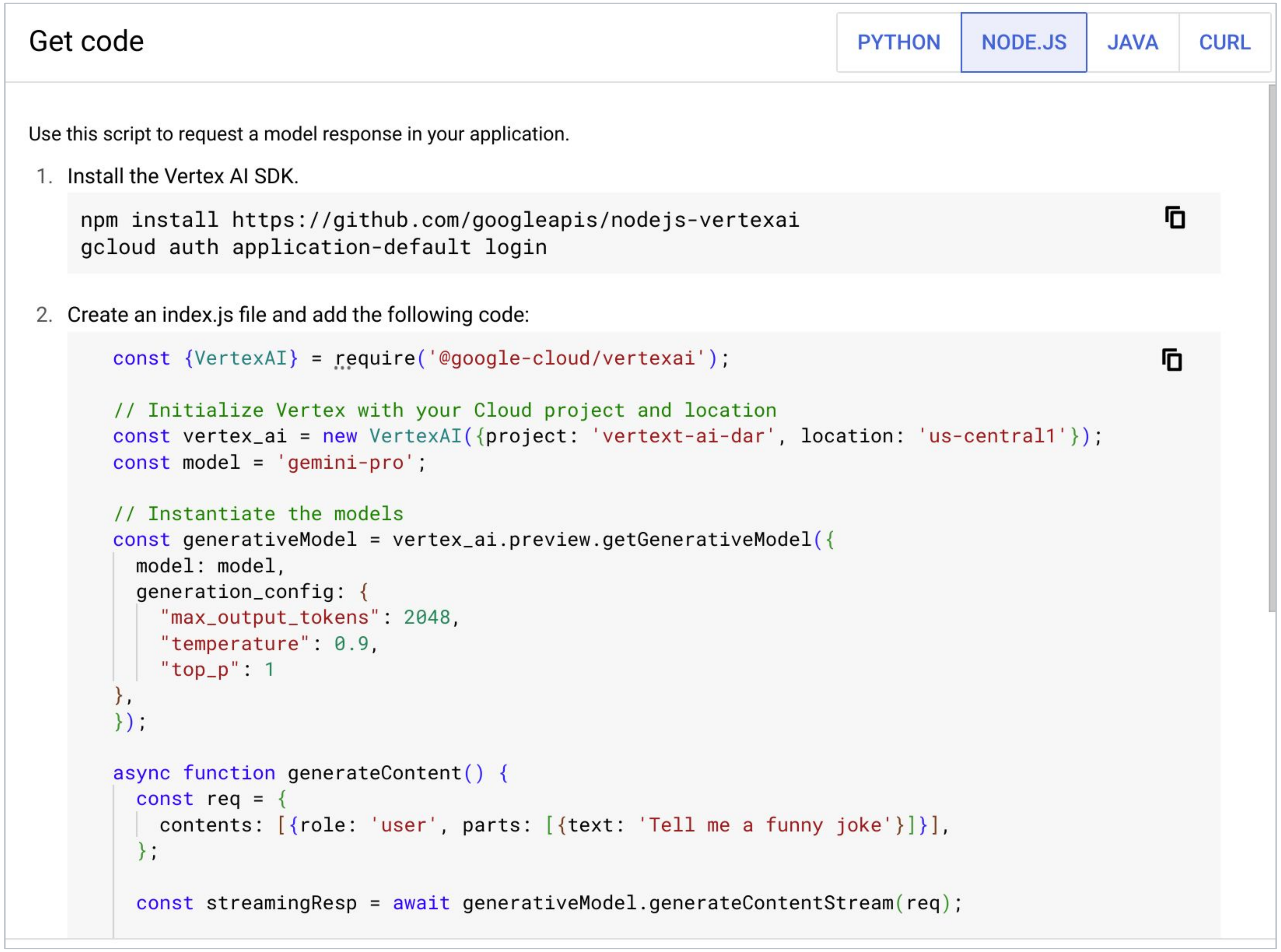
1. Set up your Java Development Environment
2. Authenticate

```
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>
```

JavaScript code example



Get code

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

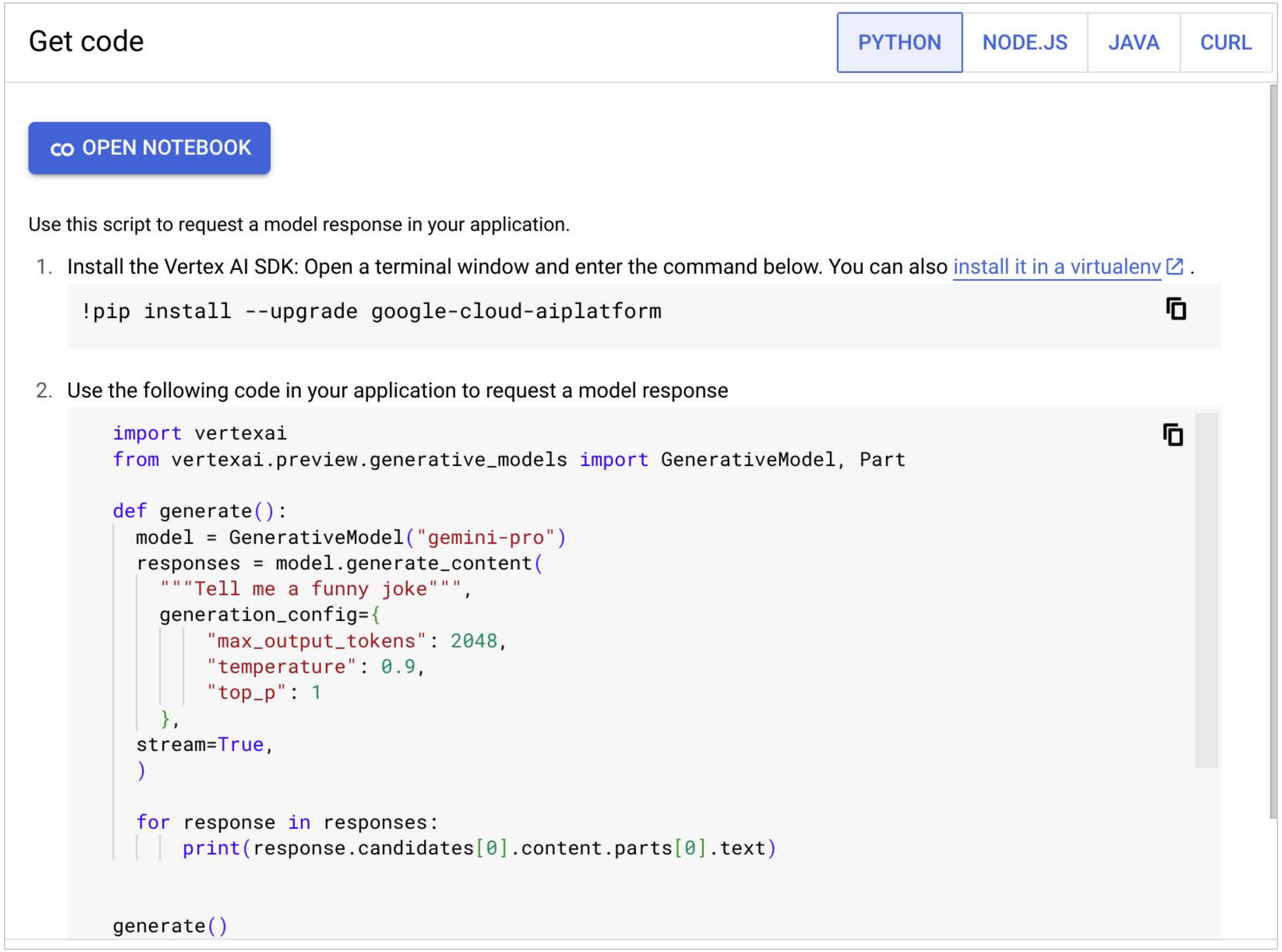
1. Install the Vertex AI SDK.

```
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);
```

Python code example



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

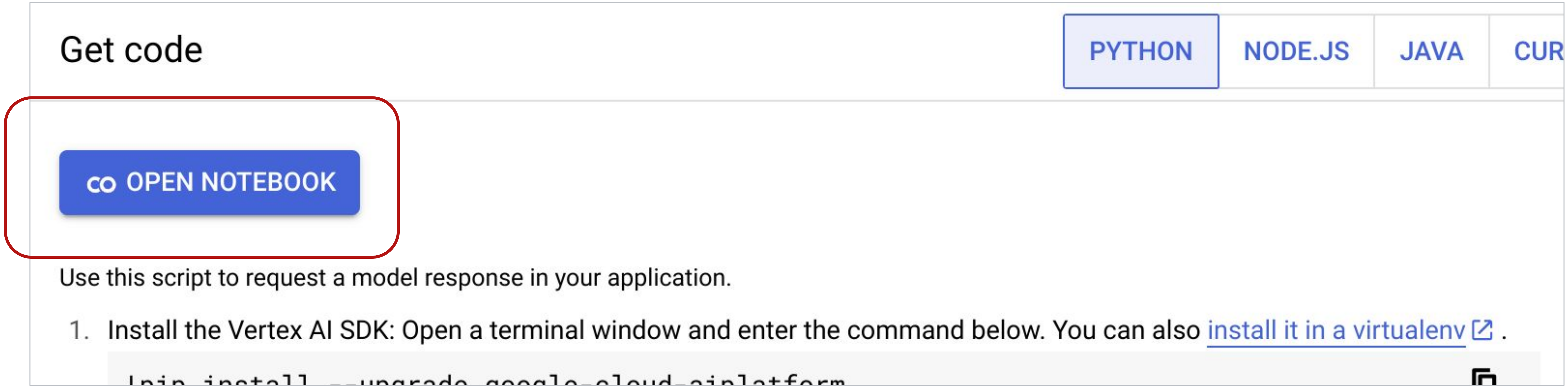
1. Install the Vertex AI SDK: Open a terminal window and enter the command below. You can also install it in a virtualenv .

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

2. Use the following code in your application to request a model response

```
import vertexai
from vertexai.preview.generative_models import GenerativeModel, Part

def generate():
  model = GenerativeModel("gemini-pro")
  responses = model.generate_content(
    """Tell me a funny joke""",
    generation_config={
        "max_output_tokens": 2048,
        "temperature": 0.9,
        "top_p": 1
    },
  stream=True,
  )

  for response in responses:
      print(response.candidates[0].content.parts[0].text)


generate()
```

Click the Open Notebook button to run the code in a Jupyter Notebook



Get code

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
```

REST API response

```
{ "predictions": [
    {
      "citationMetadata": {
        "citations": []
      },
      "content": "```\nI am a programmer,\...\n```",
      "safetyAttributes": {
        "categories": [],
        "blocked": false,
        "scores": []}}],
  "metadata": {
    "tokenMetadata": {
      "inputTokenCount": {
        "totalTokens": 8,
        "totalBillableCharacters": 33},
      "outputTokenCount": {
        "totalTokens": 130,
        "totalBillableCharacters": 355}}}}
```

Where the response was derived

The response

Input and output tokens determine the cost

Safety attributes

- Returns an array of categories and an array of scores
  - A category is only returned if its score is greater than 0
  - Score is a value between 0.0 and 1.0
  - The higher the score the more likely the content violates that category
- You should check those values before responding to a user
  - Set a threshold where responses should be blocked

```
"safetyAttributes": {
  "categories": [
      "Death, Harm & Tragedy",
      "Public Safety",
      "Religion & Belief",
      "War & Conflict"
    ],
    "blocked": false,
    "scores": [
      0.9,
      0.1,
      0.2,
      0.8
    ]
}
```

Topics

01 Getting Started

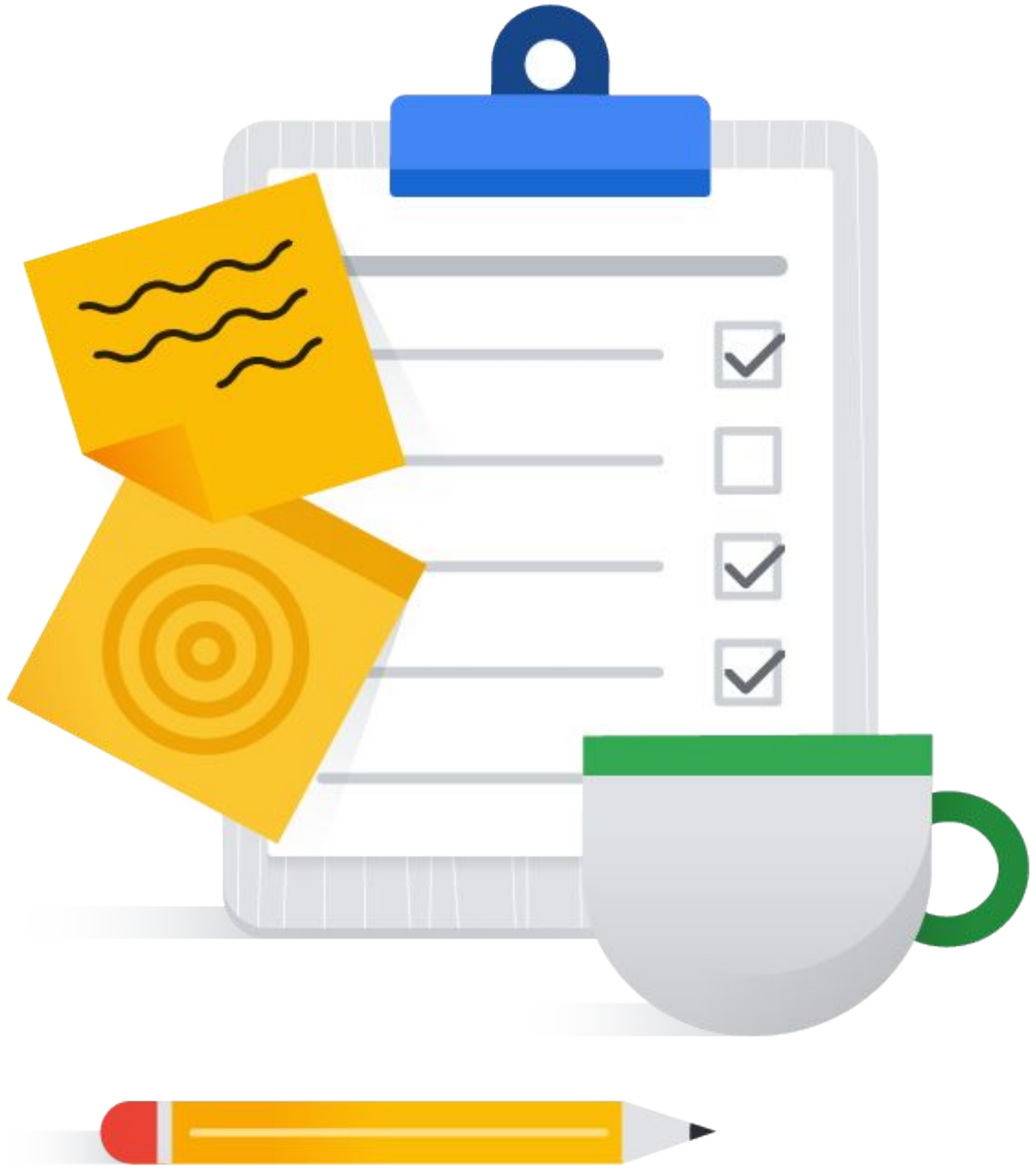
02 Python Language API

03 Programming Text Generation Applications

04 Programming Chat Applications

05 Large Document Processing

06 Imagen and Gemini



Vertex AI requirements for Python

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

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

- Import Vertex AI and the classes required for your application

```
import vertexai
from vertexai.language_models import TextGenerationModel
```

Basic Python code for Text Generation

```
import vertexai
from vertexai.language_models import TextGenerationModel
vertexai.init(project="your-proj-id", location="us-central1")
parameters = {
    "candidate_count": 1,
    "max_output_tokens": 1024,
    "temperature": 0.2,
    "top_p": 0.8,
    "top_k": 40
}
model = TextGenerationModel.from_pretrained("text-bison")
response = model.predict(
    """Tell me about Grace Hopper""", **parameters
)
print(f"Response from Model: {response.text}")
```

Set parameters appropriate to your use case

Create an instance of the model and use predict to make the request

Get the output using the text property of the response

Basic Python code for Chat

```
import vertexai
from vertexai.language_models import ChatModel, InputOutputTextPair

vertexai.init(project="your-proj-id", location="us-central1")
chat_model = ChatModel.from_pretrained("chat-bison")
parameters = {
    "candidate_count": 1,
    "max_output_tokens": 1024,
    "temperature": 0.2,
    "top_p": 0.8,
    "top_k": 40
}
chat = chat_model.start_chat()
response = chat.send_message("""Who was Steve Jobs?""", **parameters)
print(f"Response from Model: {response.text}")
```

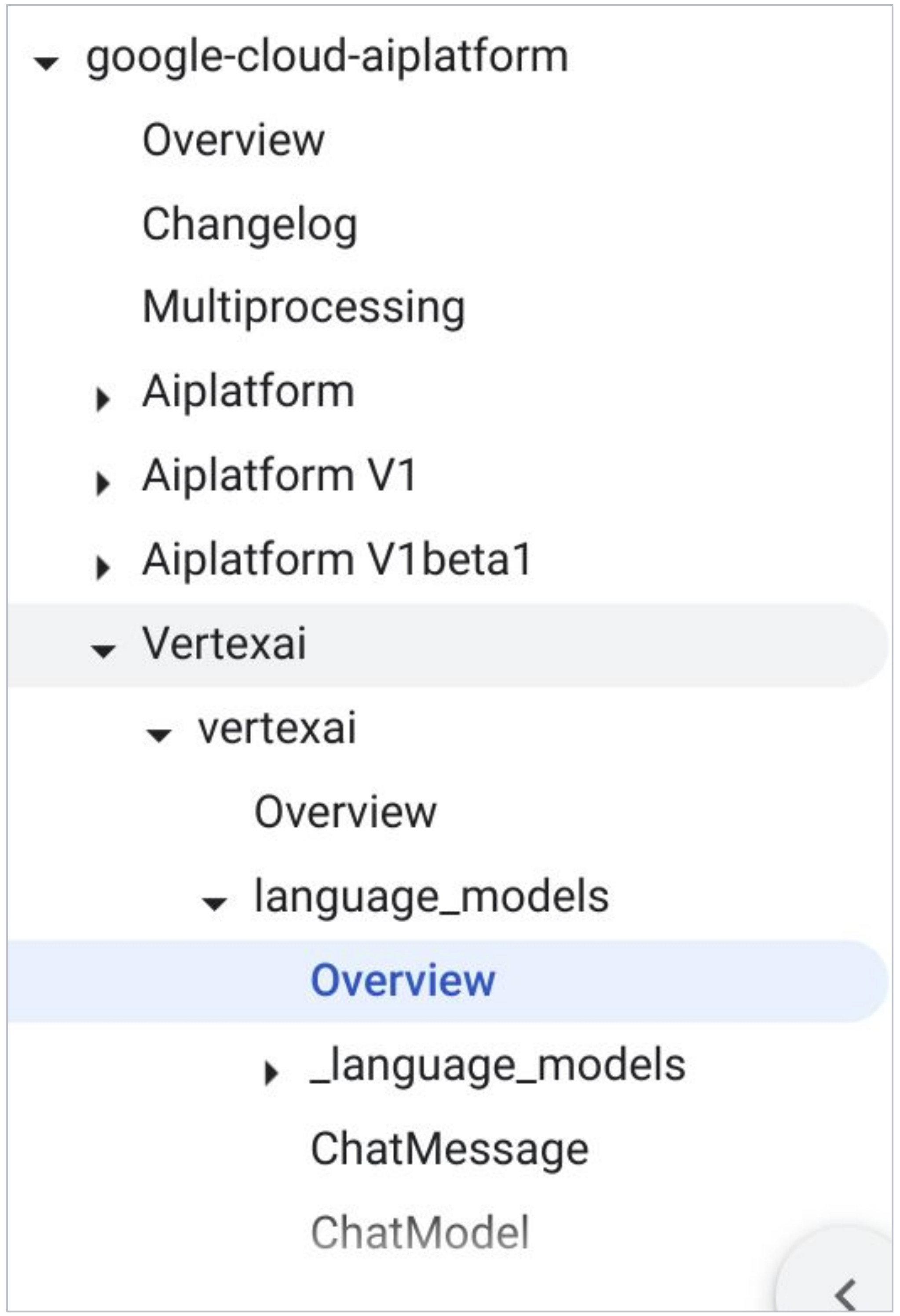
When you start a chat the history of the conversation is maintained

Request parameters

Property	Description
temperature	● Value between 0 and 1 ● Controls the degree of randomness in the output ● 0 is deterministic (it always returns the highest probable token
maxOutputTokens	● Maximum size of the response ● Value between 1 and 2048
topK	● Value between 1 and 40 ● Determines the number of tokens that can be chosen
topP	● Value between 0 and 1 ● Tokens are selected from most probable to least until the sum of their probabilities equals the top-p value.
candidateCount	● The number of candidate responses to return

Python API Documentation

- Drill down to:
  google-cloud-aiplatform > Vertexai > vertexai > language_models
- For text generation models the important classes are:
  - `TextGenerationModel`
  - `TextGenerationResponse`
- For chat uses cases the important classes are;
  - `ChatModel`
  - `ChatSession`
  - `ChatMessage`
- Examples are added using the class
  - `InputOutputTextPair`



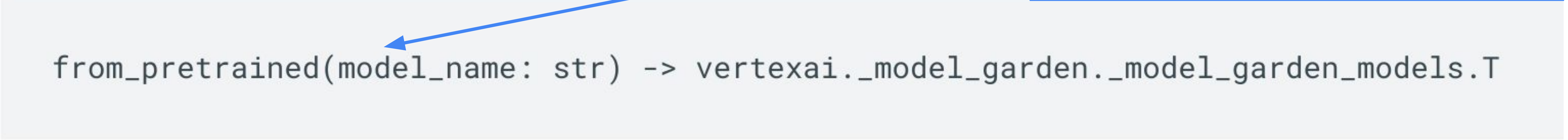
- google-cloud-aiplatform
  - Overview
  - Changelog
  - Multiprocessing
  - Aiplatform
  - Aiplatform V1
  - Aiplatform V1beta1
  - Vertexai
    - vertexai
      - Overview
      - language_models
        - Overview
        - _language_models
        - ChatMessage
        - ChatModel

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

TextGenerationModel Methods

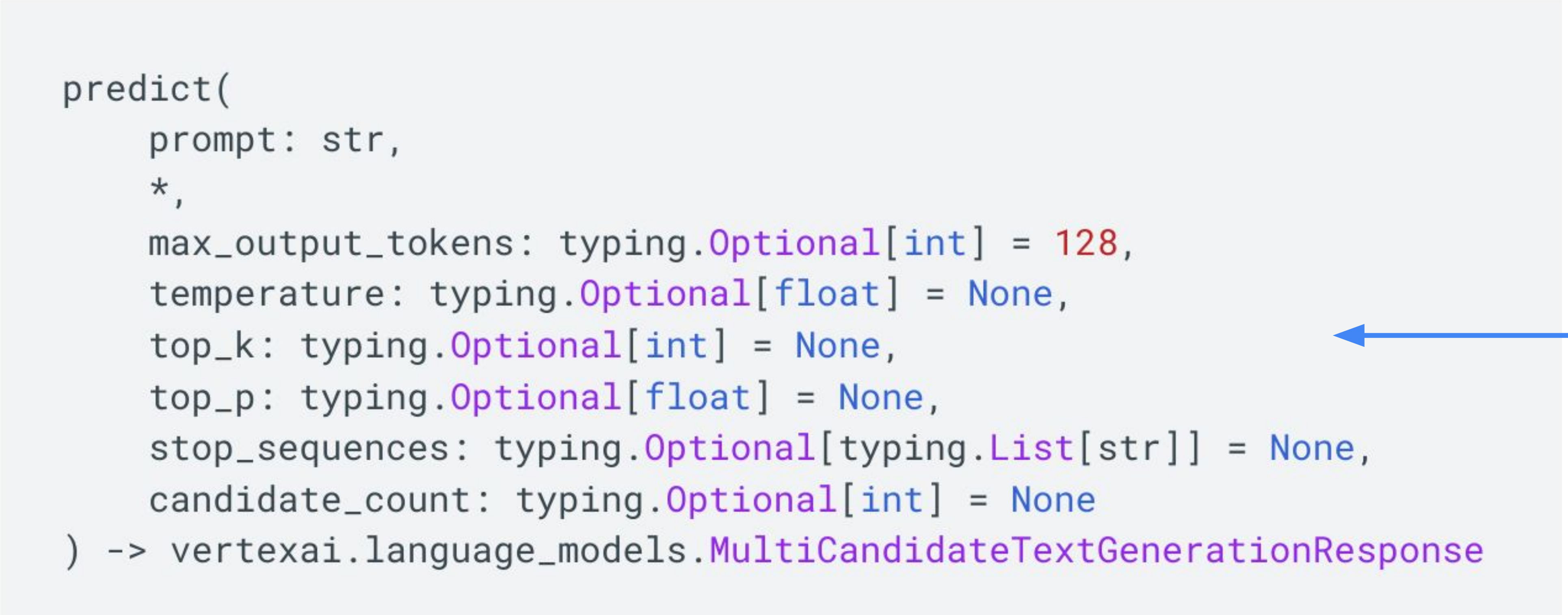
Factory method to create the model instance

from_pretrained



```
from_pretrained(model_name: str) -> vertexai._model_garden._model_garden_models.T
```

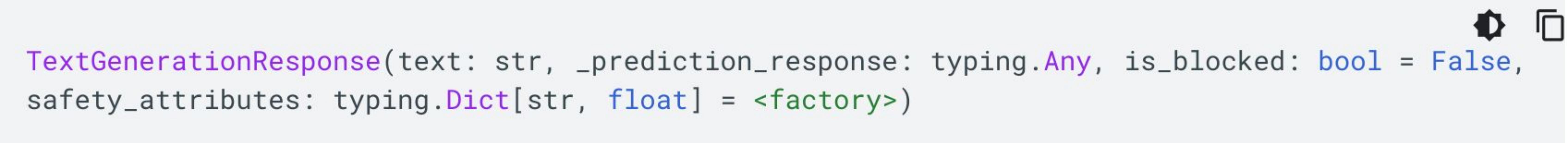
predict



```
predict(
    prompt: str,
    *,
    max_output_tokens: typing.Optional[int] = 128,
    temperature: typing.Optional[float] = None,
    top_k: typing.Optional[int] = None,
    top_p: typing.Optional[float] = None,
    stop_sequences: typing.Optional[typing.List[str]] = None,
    candidate_count: typing.Optional[int] = None
) -> vertexai.language_models.MultiCandidateTextGenerationResponse
```

Pass in the prompt and parameters to get a model response

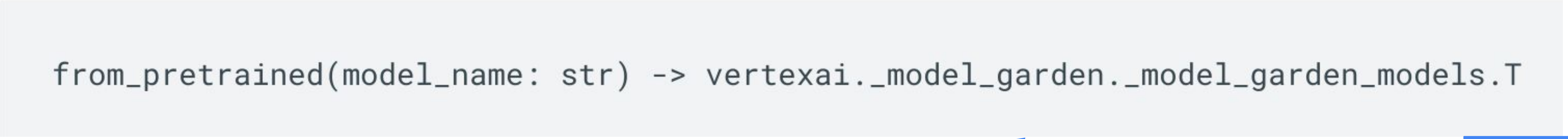
`TextGenerationResponse` contains the response from the LLM including text and safety attributes



```
TextGenerationResponse(text: str, _prediction_response: typing.Any, is_blocked: bool = False,
safety_attributes: typing.Dict[str, float] = <factory>)
```

ChatModel Methods

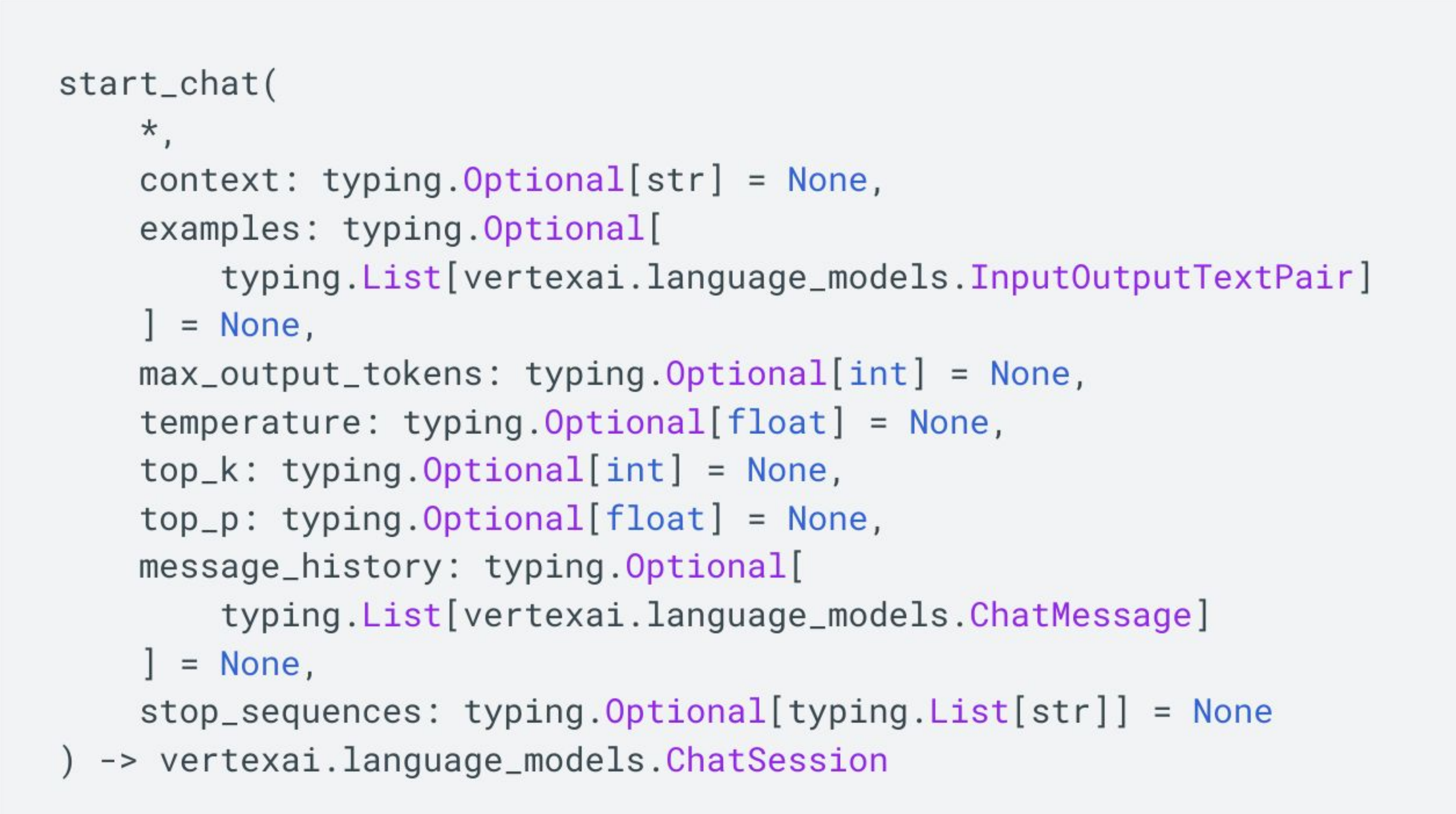
from_pretrained



```
from_pretrained(model_name: str) -> vertexai._model_garden._model_garden_models.T
```

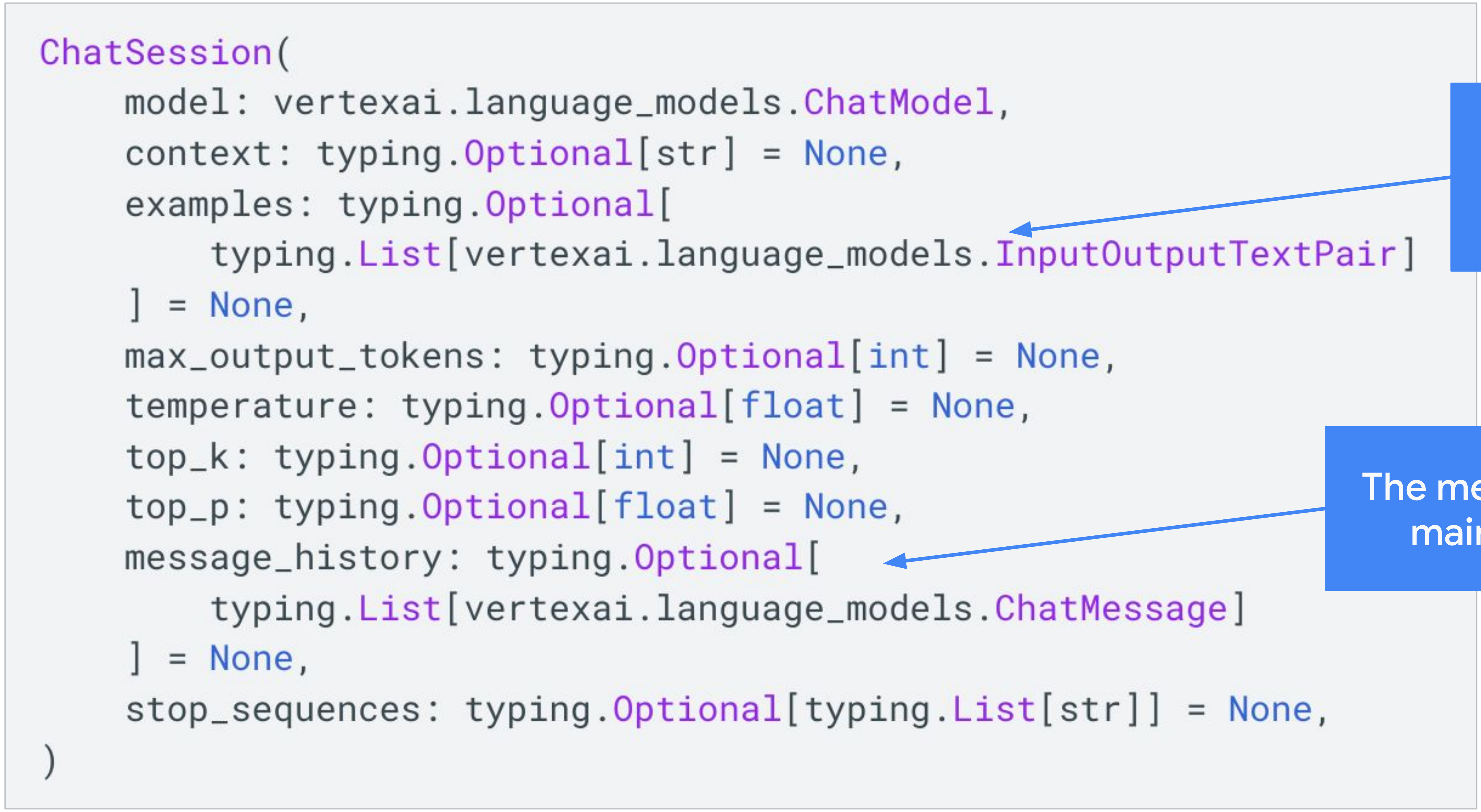
Factory method to create the model instance

start_chat



```
start_chat(
    *,
    context: typing.Optional[str] = None,
    examples: typing.Optional[
        typing.List[vertexai.language_models.InputOutputTextPair]
    ] = None,
    max_output_tokens: typing.Optional[int] = None,
    temperature: typing.Optional[float] = None,
    top_k: typing.Optional[int] = None,
    top_p: typing.Optional[float] = None,
    message_history: typing.Optional[
        typing.List[vertexai.language_models.ChatMessage]
    ] = None,
    stop_sequences: typing.Optional[typing.List[str]] = None
) -> vertexai.language_models.ChatSession
```

When you call the `start_chat` function a `ChatSession` is created



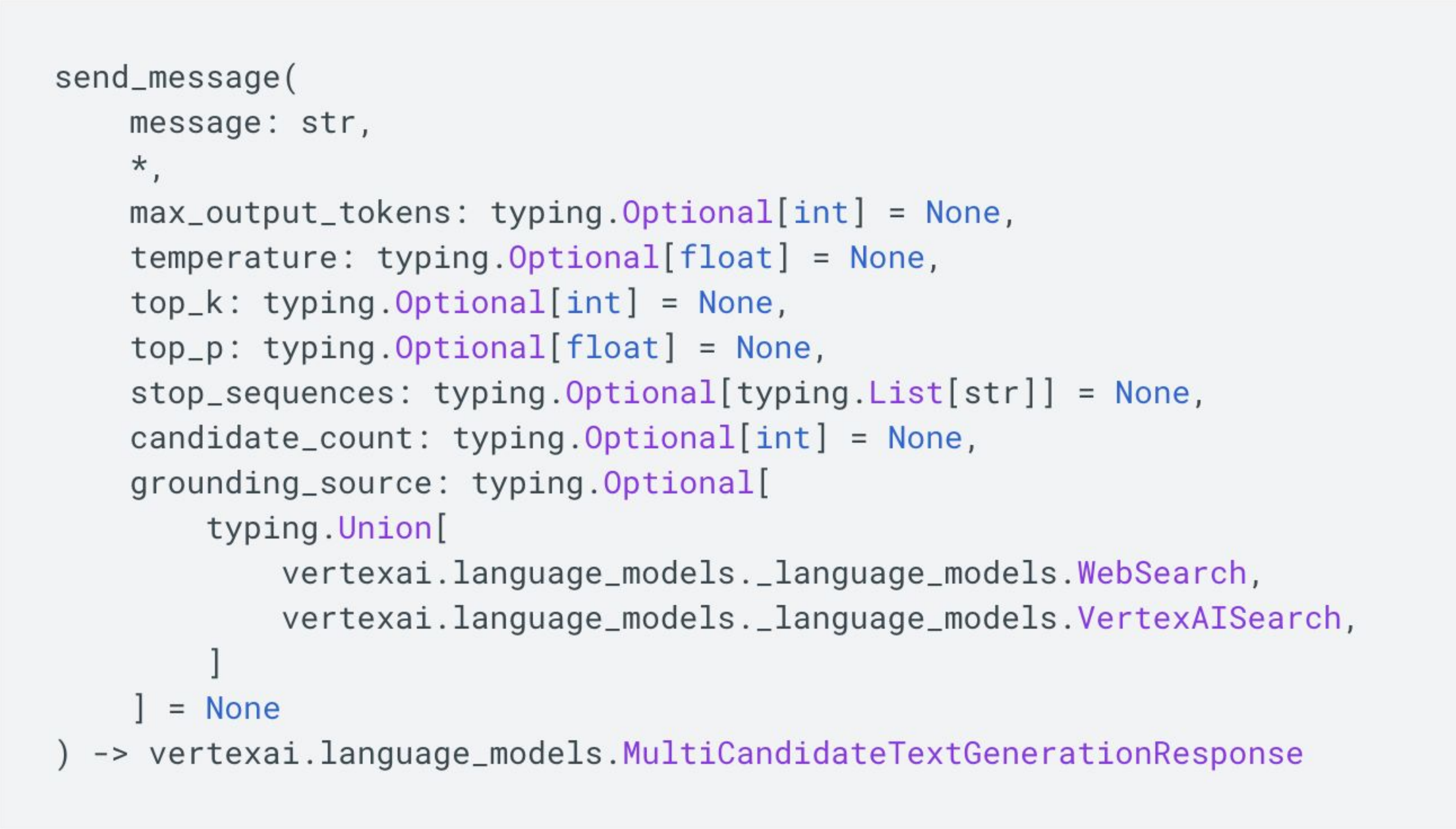
```
ChatSession(
    model: vertexai.language_models.ChatModel,
    context: typing.Optional[str] = None,
    examples: typing.Optional[
        typing.List[vertexai.language_models.InputOutputTextPair]
    ] = None,
    max_output_tokens: typing.Optional[int] = None,
    temperature: typing.Optional[float] = None,
    top_k: typing.Optional[int] = None,
    top_p: typing.Optional[float] = None,
    message_history: typing.Optional[
        typing.List[vertexai.language_models.ChatMessage]
    ] = None,
    stop_sequences: typing.Optional[typing.List[str]] = None,
)
```

Context, examples, and parameters are maintained within the `ChatSession`

The message history has to be maintained for each user

ChatSession Methods

send_message



```
send_message(
    message: str,
    *,
    max_output_tokens: typing.Optional[int] = None,
    temperature: typing.Optional[float] = None,
    top_k: typing.Optional[int] = None,
    top_p: typing.Optional[float] = None,
    stop_sequences: typing.Optional[typing.List[str]] = None,
    candidate_count: typing.Optional[int] = None,
    grounding_source: typing.Optional[
        typing.Union[
            vertexai.language_models._language_models.WebSearch,
            vertexai.language_models._language_models.VertexAISearch,
        ]
    ] = None
) -> vertexai.language_models.MultiCandidateTextGenerationResponse
```

The ChatSession `message_history` property is a collection of `ChatMessage` objects

```
ChatMessage(content: str, author: str)
```

Examples are added using `InputOutputTextPair` objects

```
InputOutputTextPair(input_text: str, output_text: str)
```

Streaming responses can make the user interface more responsive

- Streaming with text generation use the `predict_streaming` function
  - Returns a collection of responses

```
responses = model.predict_streaming("Tell me about Steve Jobs",**parameters)

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

- Streaming with a chat session use the `send_message_streaming` function

```
responses = chat.send_message_streaming("Tell me about Grace Hopper", **parameters)

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

Topics

01 Getting Started

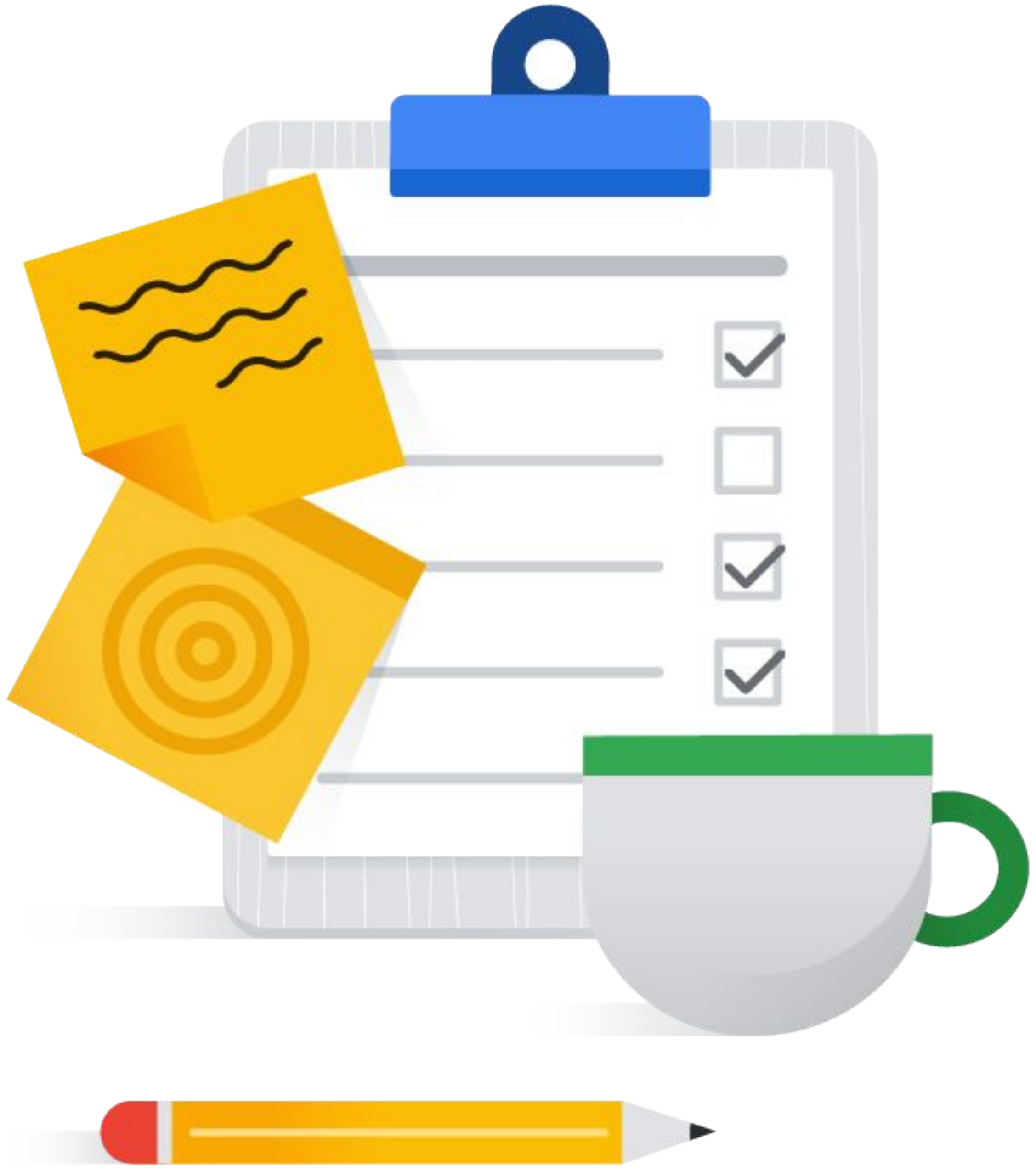
02 Python Language API

03 Programming Text Generation Applications

04 Programming Chat Applications

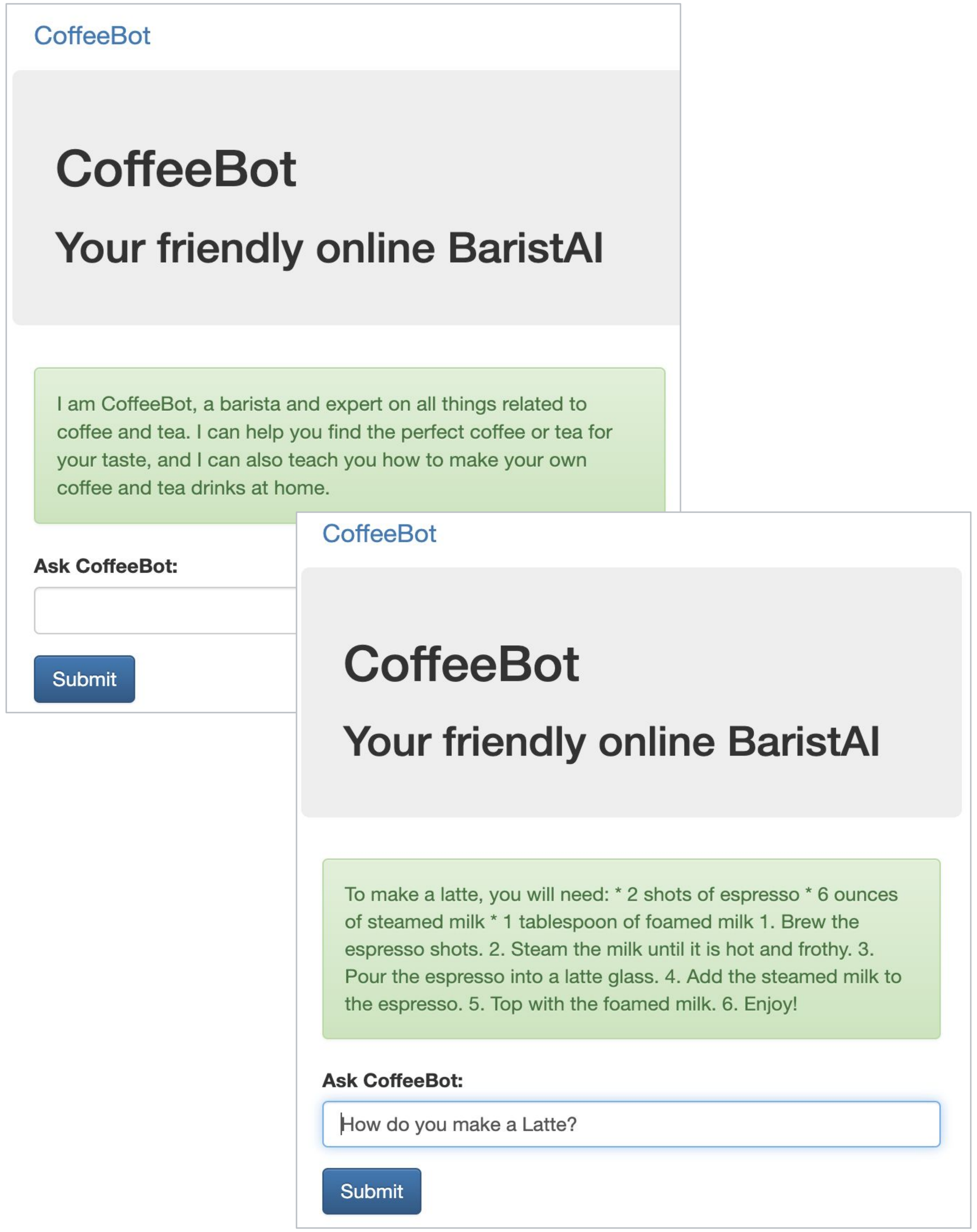
05 Large Document Processing

06 Imagen and Gemini



Python Flask Website example

- This is an example of using the text service with the PaLM API
  - Even though you may ask many questions, each one is independent
- Context must be added to tell the PaLM API to emulate a barista
- The coding is simple as you are just submitting an HTML form and making a request to the PaLM API for a response
  - The response is displayed on the screen



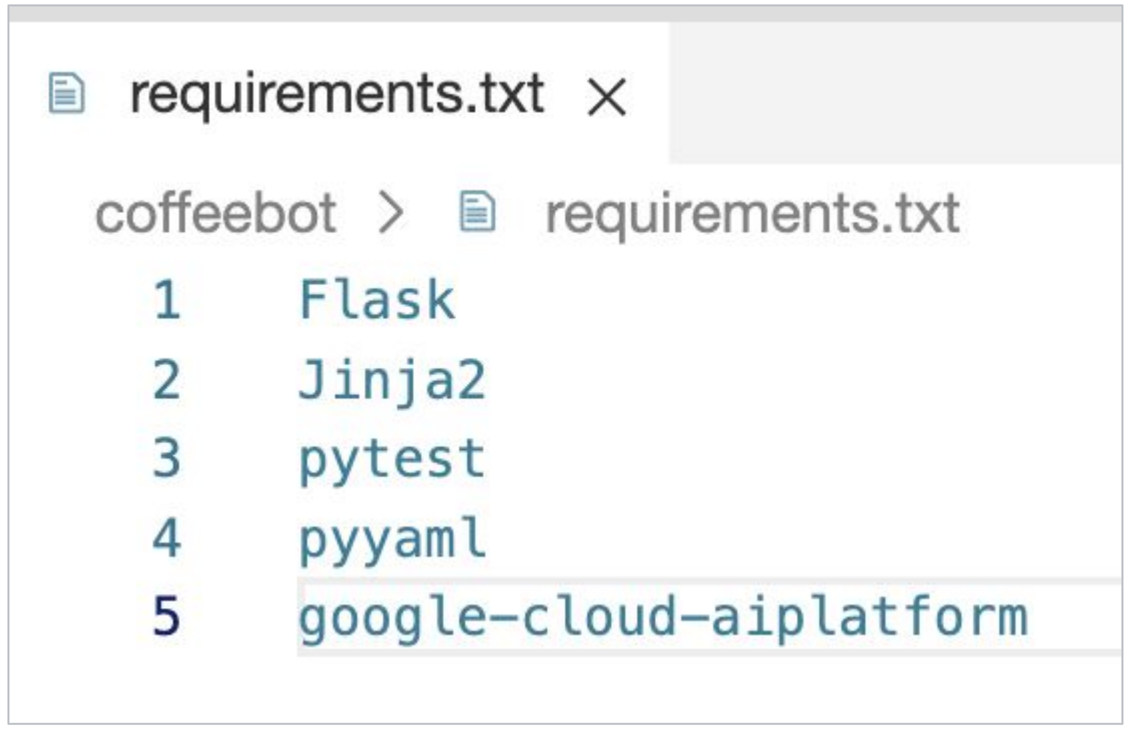
I am CoffeeBot, a barista and expert on all things related to coffee and tea. I can help you find the perfect coffee or tea for your taste, and I can also teach you how to make your own coffee and tea drinks at home.

To make a latte, you will need: * 2 shots of espresso * 6 ounces of steamed milk * 1 tablespoon of foamed milk 1. Brew the espresso shots. 2. Steam the milk until it is hot and frothy. 3. Pour the espresso into a latte glass. 4. Add the steamed milk to the espresso. 5. Top with the foamed milk. 6. Enjoy!

Ask CoffeeBot: How do you make a Latte?

Add the Python requirements

- Add Google Cloud AI Platform to the requirements.txt file
- Add the required imports at the top of the code file



```
Flask
Jinja2
pytest
pyyaml
google-cloud-aiplatform
```

```
from flask import Flask, render_template, request
import os
import vertexai
from vertexai.language_models import TextGenerationModel
```

Handling web requests in Flask

- The default route will handle HTTP posts and gets
  - Post means a question was submitted from the HTML form
  - Get means there is no question (have CoffeeBot introduce itself)
- The code for using the PaLM API is in the `get_response()` function

```
@app.route("/", methods = ['POST', 'GET'])
def main():
    if request.method == 'POST':
        input = request.form['input']
        response = get_response(input)
    else:
        input = ""
        response = get_response("Who are you and what can you do?")

    model = {"title": "CoffeeBot", "message": response, "input": input}
    return render_template('index.html', model=model)
```

Making a request to the PaLM API

```
def get_response(input):
    vertexai.init(project="vertext-ai-dar", location="us-central1")
    parameters = {
        "temperature": 0.8,
        "max_output_tokens": 256,
        "top_p": 0.8,
        "top_k": 40
    }

    model = TextGenerationModel.from_pretrained("text-bison@001")

                << CODE CONTINUED ON NEXT SLIDE >>
```

Initialize the API and set up the parameters

Create the model using the correct version of the PaLM API

Making a request to the PaLM API (continued)

```
def get_response(input):
                 << CODE CONTINUED FROM PREVIOUS SLIDE >>

    model = TextGenerationModel.from_pretrained("text-bison@001")
    request = """Your name is CoffeeBot. You are a barista and expert on
    all things related to coffee and tea..

    input: {}
    output:
    """
    response = model.predict(
        request.format(input),
        **parameters
    )
    return response
```

The context tells the API to emulate a barista

The input is the question

Call the `model.predict()` function to send the request

The format function injects the question into the prompt

Topics

01 Getting Started

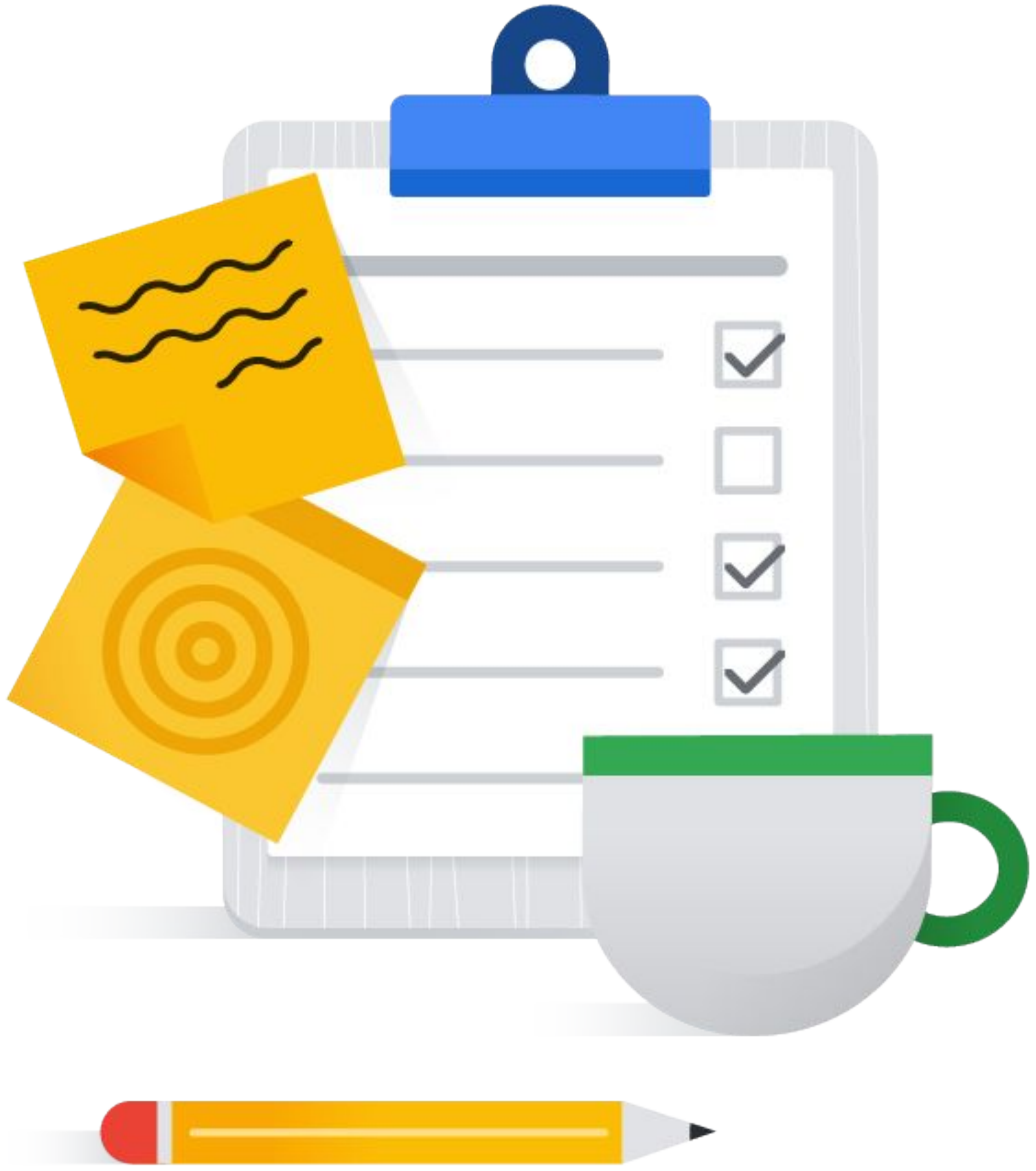
02 Python Language API

03 Programming Text Generation Applications

04 Programming Chat Applications

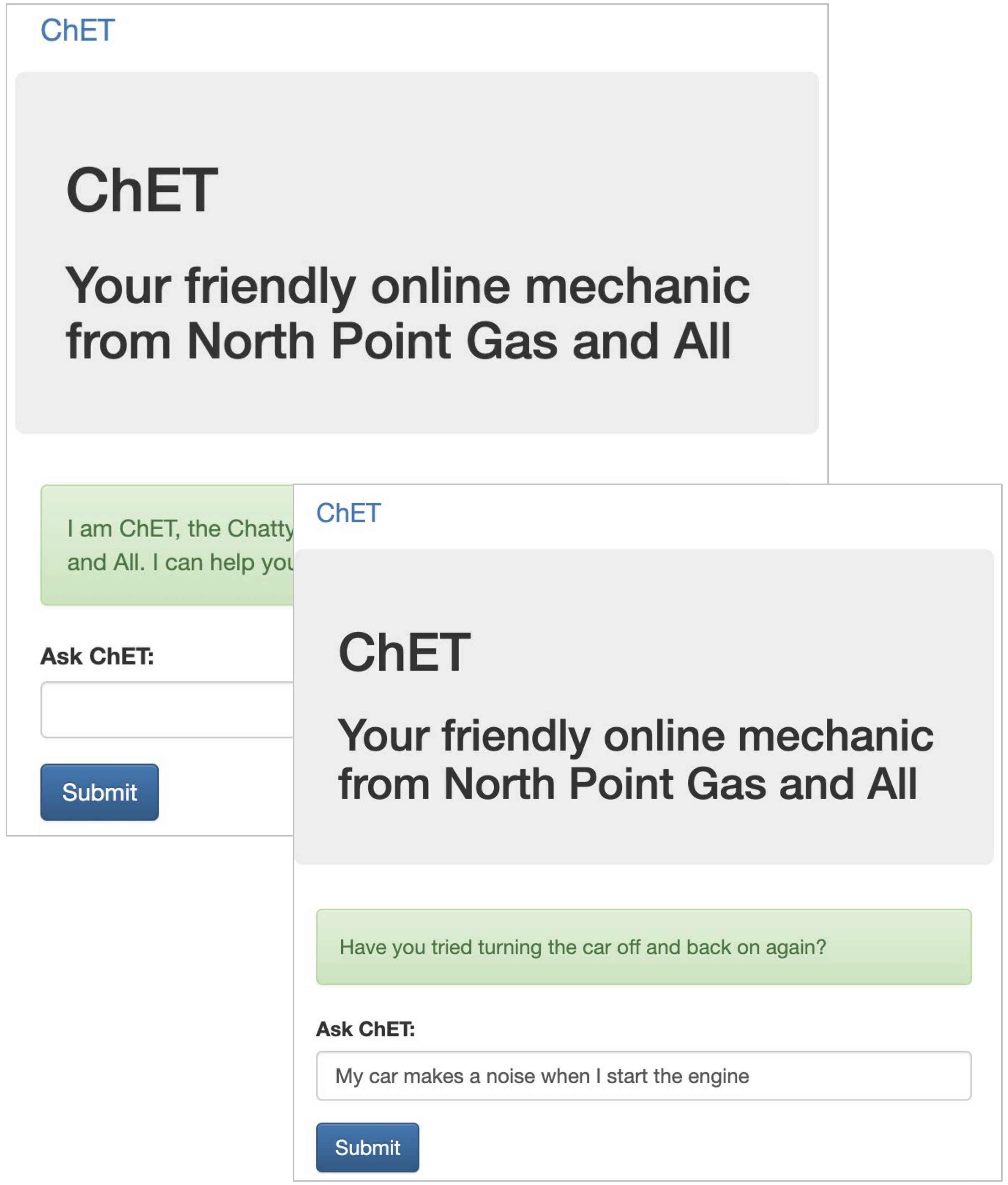
05 Large Document Processing

06 Imagen and Gemini



Python Flask Website example

- This is an example of using the chat service with the PaLM API
  - The chat will remember the history of the conversation
- Context must be added to tell the PaLM API to emulate a customer service agent for the service station
- The coding is simple as you are just submitting an HTML form and making a request to the PaLM API for a response
  - The response is displayed on the screen



ChET

ChET

Your friendly online mechanic from North Point Gas and All

I am ChET, the Chatty
and All. I can help you

Ask ChET:

Submit

ChET

ChET

Your friendly online mechanic from North Point Gas and All

Have you tried turning the car off and back on again?

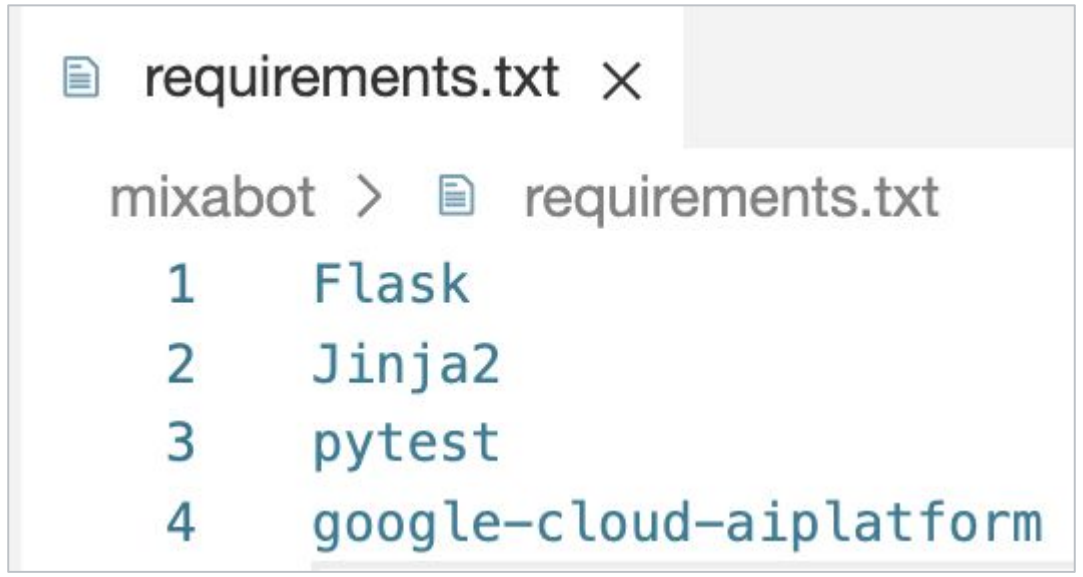
Ask ChET:

My car makes a noise when I start the engine

Submit

Add the Python requirements

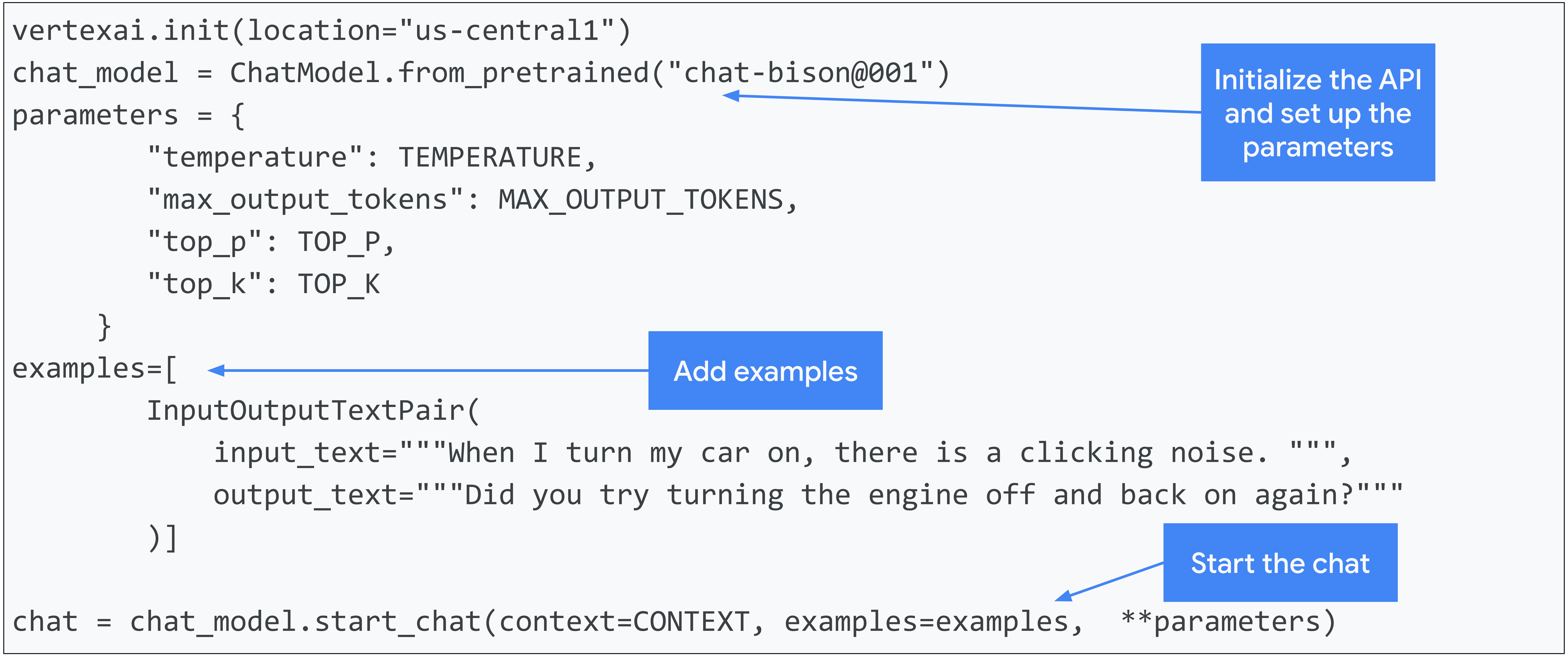
- Add Google Cloud AI Platform to the `requirements.txt` file
- Add the required imports at the top of the code file



```
Flask
Jinja2
pytest
google-cloud-aiplatform
```

```
from flask import Flask, render_template, request
import os
import vertexai
From vertexai.preview.language_models import ChatModel, InputOutputTextPair,
ChatMessage
```

Initializing the Chat session



```
vertexai.init(location="us-central1")
chat_model = ChatModel.from_pretrained("chat-bison@001")
parameters = {
        "temperature": TEMPERATURE,
        "max_output_tokens": MAX_OUTPUT_TOKENS,
        "top_p": TOP_P,
        "top_k": TOP_K
     }
examples=[
        InputOutputTextPair(
            input_text="""When I turn my car on, there is a clicking noise. """,
            output_text="""Did you try turning the engine off and back on again?"""
        )]

chat = chat_model.start_chat(context=CONTEXT, examples=examples,  **parameters)
```

Initialize the API and set up the parameters

Add examples

Start the chat

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
```

```
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
```

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

Topics

01 Getting Started

02 Python Language API

03 Programming Text Generation Applications

04 Programming Chat Applications

05 Large Document Processing

06 Imagen and Gemini

What are tokens?

- When text is sent to an LLM it is split into tokens
  - A token represents an idea that the large language model understands
- Tokens are on average about 4-5 characters long
  - A short word may be converted into a single token
  - Large words might use multiple tokens
  - Punctuation is represented as tokens
- When they are being processed, tokens are converted into numeric arrays called embeddings
- The model generates output embeddings which are converted back into tokens and returned to the caller
- A token is approximately four characters (100 tokens correspond to roughly 60-80 words)

There is a limit to the number of input and output tokens that can be processed by the model

- Limits change based on model version and will likely increase over time
  - The `text-bison-32k` and `chat-bison-32k` support 32,000 tokens in the request-response
- If you want the model to summarize documents that exceed the token limit you need to split the operation into multiple calls
  - Like a map-reduce operation
  - Divide the document into pieces
  - Summarize each piece
  - Summarize the summaries
- With chat uses cases the entire conversation is sent with each request
  - Be careful that the conversation doesn’t exceed the limit
  - You may need to trim the history at some point
  - Alternatively, you can have the LLM summarize the conversation thus far, then send only the summary as context

Summarizing documents is a common task for large language models

- This seems simple enough:
  - Retrieve the document to be summarized
  - Add the document to the prompt
  - Ask the LLM to summarize it
- Problems:
  - What if you have many small documents to summarize?
  - What if the data is too large for a single request?
  - What happens if you exceed the quota of requests?

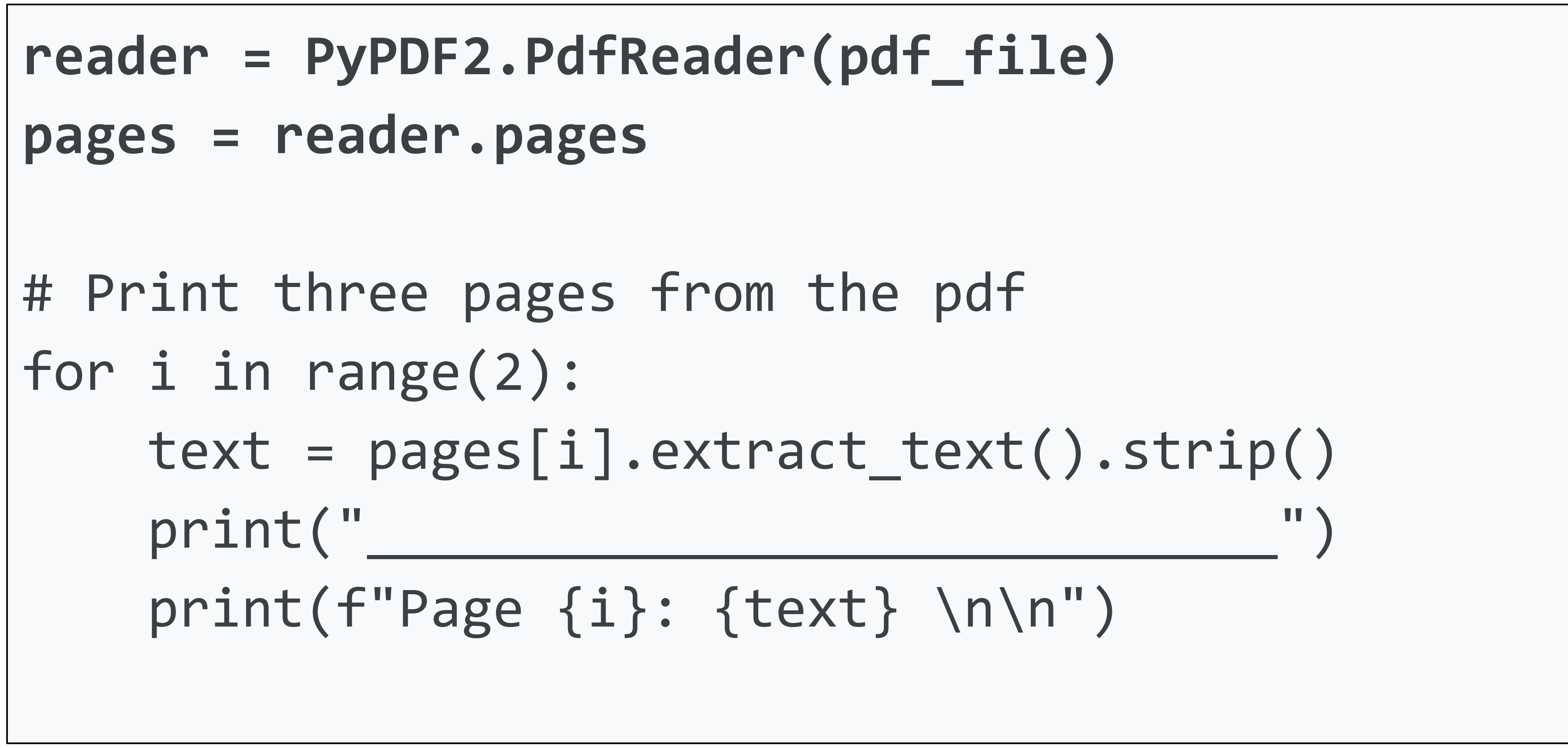
Retrieving external data

- In the example below, the Python `urllib` package is used to download a pdf file and store it locally in a folder “`data`”

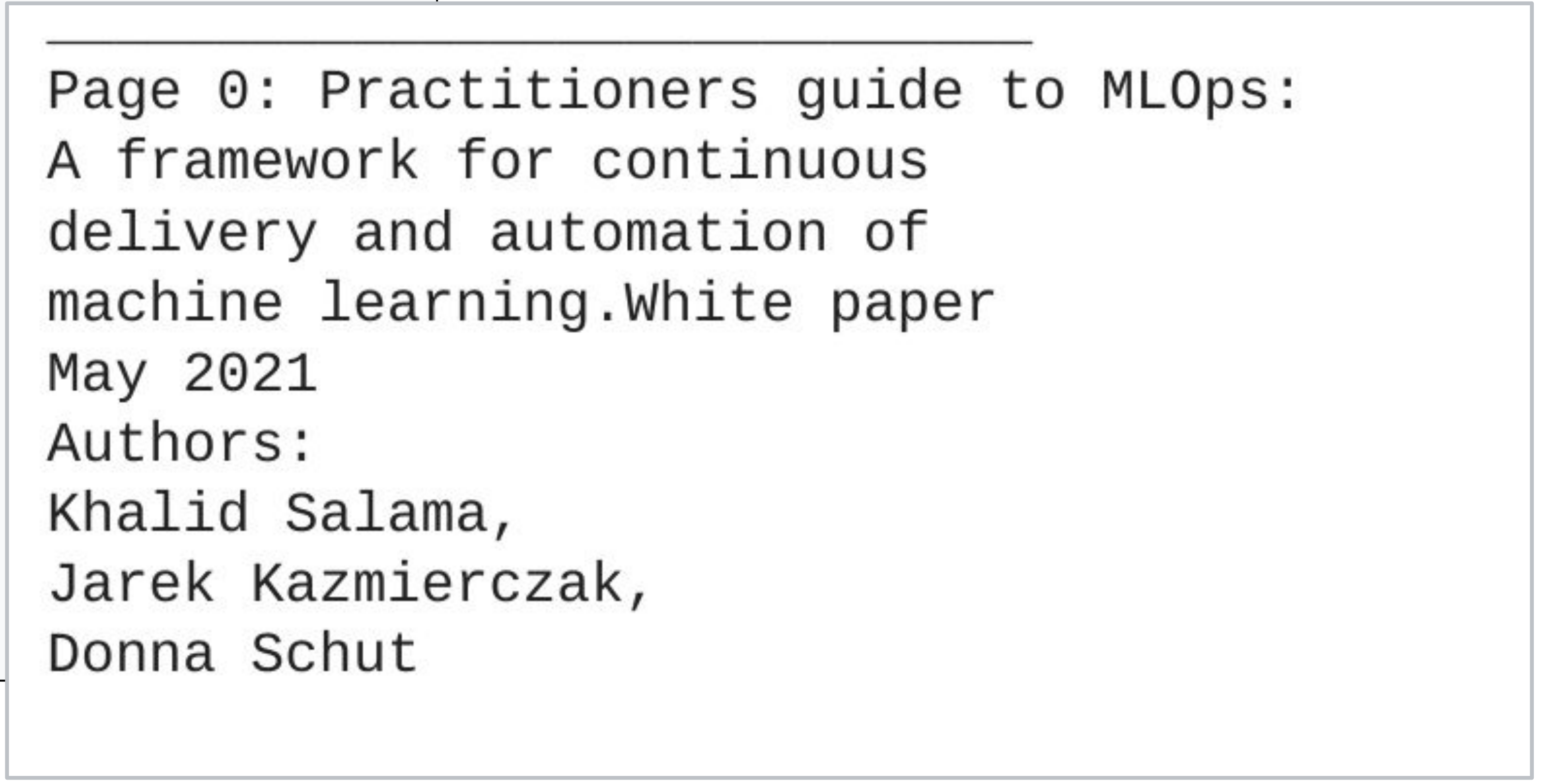
```
data_folder = "data"
Path(data_folder).mkdir(parents=True, exist_ok=True)

pdf_url =
"https://services.google.com/fh/files/misc/practitioners_guide_to_mlops_whitepaper.pdf"
pdf_file = Path(data_folder, pdf_url.split("/")[-1])

urllib.request.urlretrieve(pdf_url, pdf_file)
```

Processing the document

- Here, the document is divided into pages



```
reader = PyPDF2.PdfReader(pdf_file)
pages = reader.pages

# Print three pages from the pdf
for i in range(2):
    text = pages[i].extract_text().strip()
    print("_____________________________")
    print(f"Page {i}: {text} \n\n")
```



```
___________________________
Page 0: Practitioners guide to MLOps:
A framework for continuous
delivery and automation of
machine learning.White paper
May 2021
Authors:
Khalid Salama,
Jarek Kazmierczak,
Donna Schut
```

Stuffing means you are combining content from multiple documents or pages

- By combining documents you can reduce the number of calls to the model
  - You also can get a single summary of more than 1 thing
- In the example below, the text from all the pages is concatenated

```
reader = PyPDF2.PdfReader(pdf_file)
pages = reader.pages
concatenated_text = ""

for page in tqdm(pages):
    text = page.extract_text().strip()
    concatenated_text += text
```

In the example below, the first 19,000 words are passed to the model for summarization

- In this case, the entire document cannot be processed in one request

~~~
prompt_template = """
    Write a concise summary of the following text delimited by triple backquotes.
    Return your response in bullet points which covers the key points of the text.
    ```{text}```
    BULLET POINT SUMMARY:
"""
prompt = prompt_template.format(text=concatenated_text[:19000])

# Use the model to summarize the text using the prompt
summary = generation_model.predict(prompt=prompt, max_output_tokens=1024).text
~~~

The `MapReduce` algorithm can be used with large docs

- Steps:
  - Divide the document into chunks
  - Summarize the chunks (Map)
  - Combine the summaries
  - Summarize the summaries (Reduce)

The Map step summarizes each page and adds each summary to a collection

```
reader = PyPDF2.PdfReader(pdf_file)
pages = reader.pages

initial_summary = []
for page in tqdm(pages):
    text = page.extract_text().strip()
    prompt = initial_prompt_template.format(text=text)
    summary = model_with_limit_and_backoff(prompt=prompt, max_output_tokens=1024).text
    initial_summary.append(summary)
```

In the Reduce step, combine all the summaries and summarize those

- Be careful that the combined summaries don’t exceed the the maximum length of a request

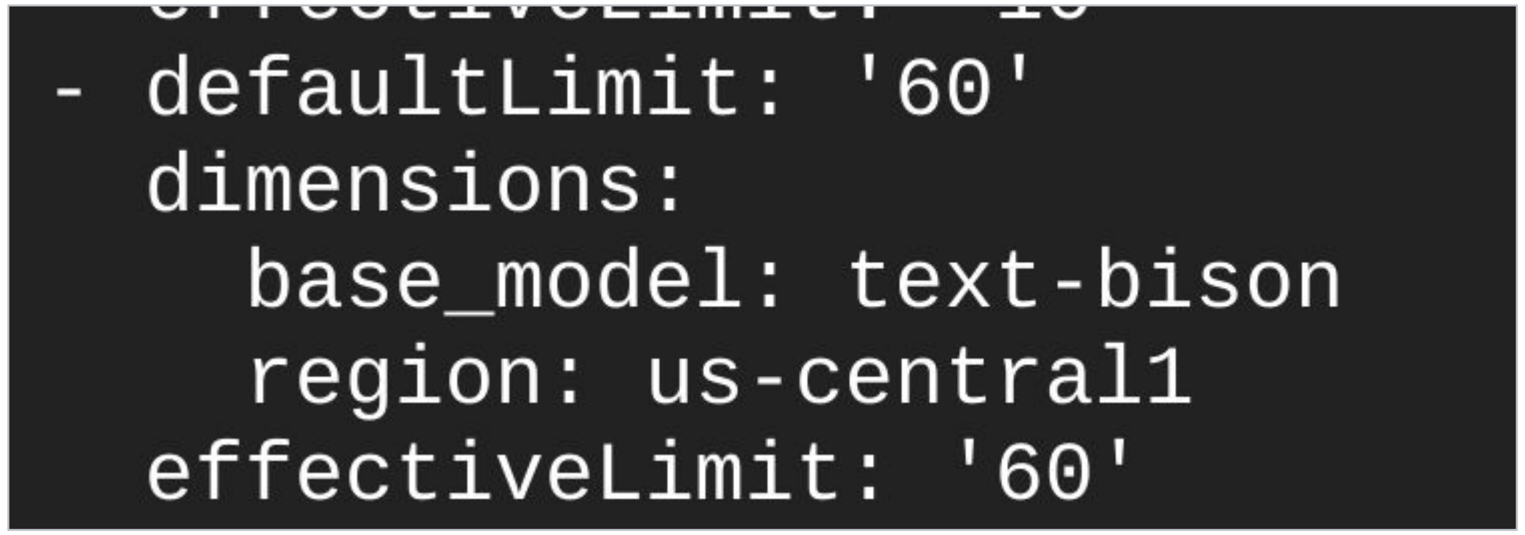
```
def reduce(initial_summary, prompt_template):
    # Concatenate the summaries from the initial step
    concat_summary = "\n".join(initial_summary)

    prompt = prompt_template.format(text=concat_summary)
    summary = model_with_limit_and_backoff(prompt=prompt, max_output_tokens=1024).text
    return summary

summary = reduce(initial_summary, final_prompt_template)
```

Notice, the call to the LLM is being made through a helper function that limits the frequency of requests

MapReduce introduces a rate limiting problem

- Quotas vary by model, project, and region
  - Likely, quotas will change over time
- Run the following gcloud command to see your quota by model

```
gcloud alpha services quota list --service=aiplatform.googleapis.com
--consumer=projects/vertext-ai-dar
--filter=metric=aiplatform.googleapis.com/online_prediction_requests_per_base_model
```



```
  effectiveLimit: '10'
- defaultLimit: '60'
  dimensions:
    base_model: text-bison
    region: us-central1
  effectiveLimit: '60'
```

Rate limiting code

```
CALL_LIMIT = 20  # Number of calls to allow within a period
ONE_MINUTE = 60  # One minute in seconds
FIVE_MINUTE = 5 * ONE_MINUTE

# A function to print a message when the function is retrying
def backoff_hdlr(details):
    print(
        "Backing off {} seconds after {} tries".format(
            details["wait"], details["tries"]
        )
    )

                               ## Continued on next slide ##
```

Rate limiting code (continued)

```
@backoff.on_exception(  # Retry with exponential backoff strategy when exceptions occur
    backoff.expo,
    (
        exceptions.ResourceExhausted,
        ratelimit.RateLimitException,
    ),  # Exceptions to retry on
    max_time=FIVE_MINUTE,
    on_backoff=backoff_hdlr,  # Function to call when retrying)

@ratelimit.limits(  # Limit the number of calls to the model per minute
    calls=CALL_LIMIT, period=ONE_MINUTE)

# This function will call the `generation_model.predict` function, but it will retry if
defined exceptions occur.
def model_with_limit_and_backoff(**kwargs):
    return generation_model.predict(**kwargs)
```

Topics

01	Getting Started
02	Python Language API
03	Programming Text Generation Applications
04	Programming Chat Applications
05	Large Document Processing
06	Imagen and Gemini

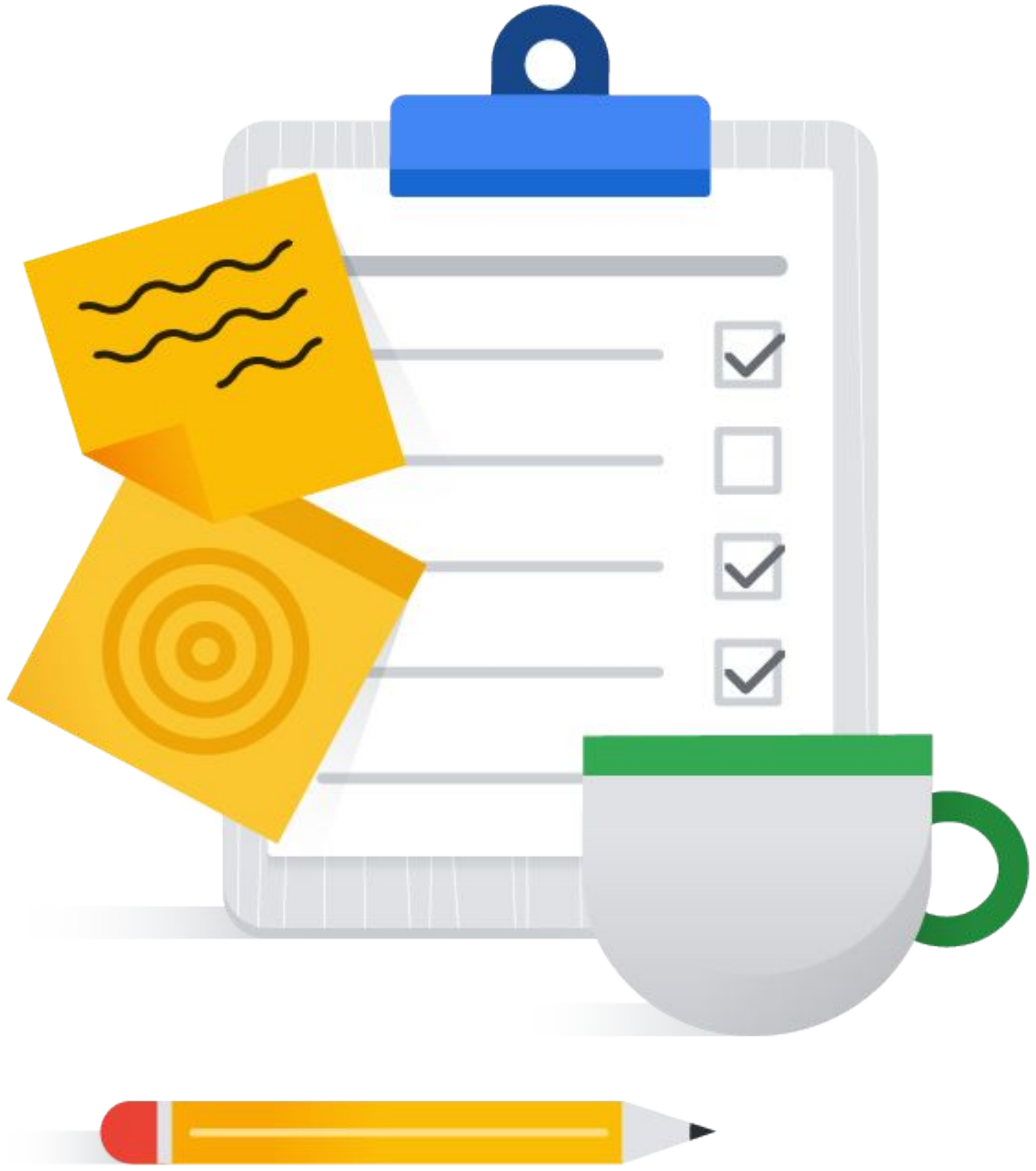
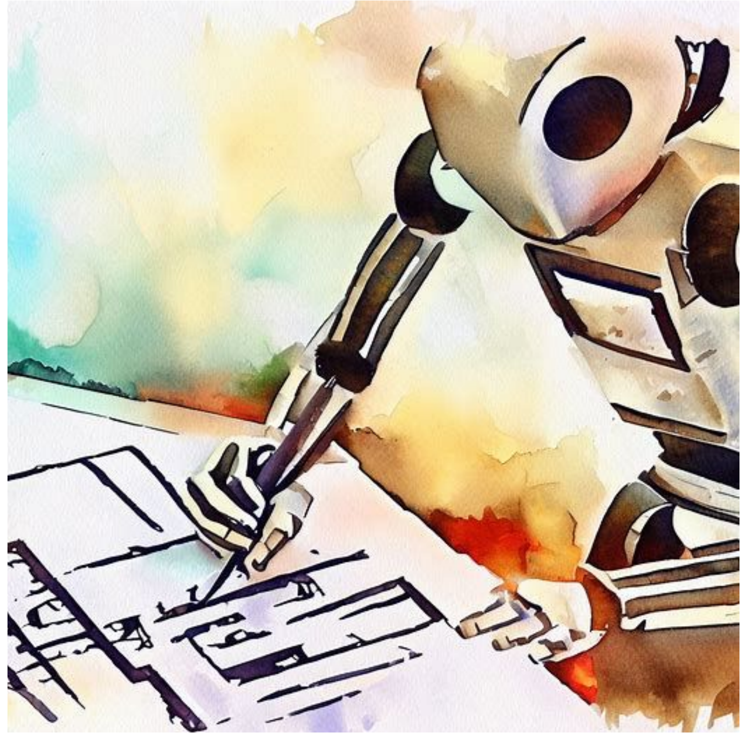


Imagen is Google’s foundation model for computer vision tasks

Image generation



Visual Q&A
Image Captioning

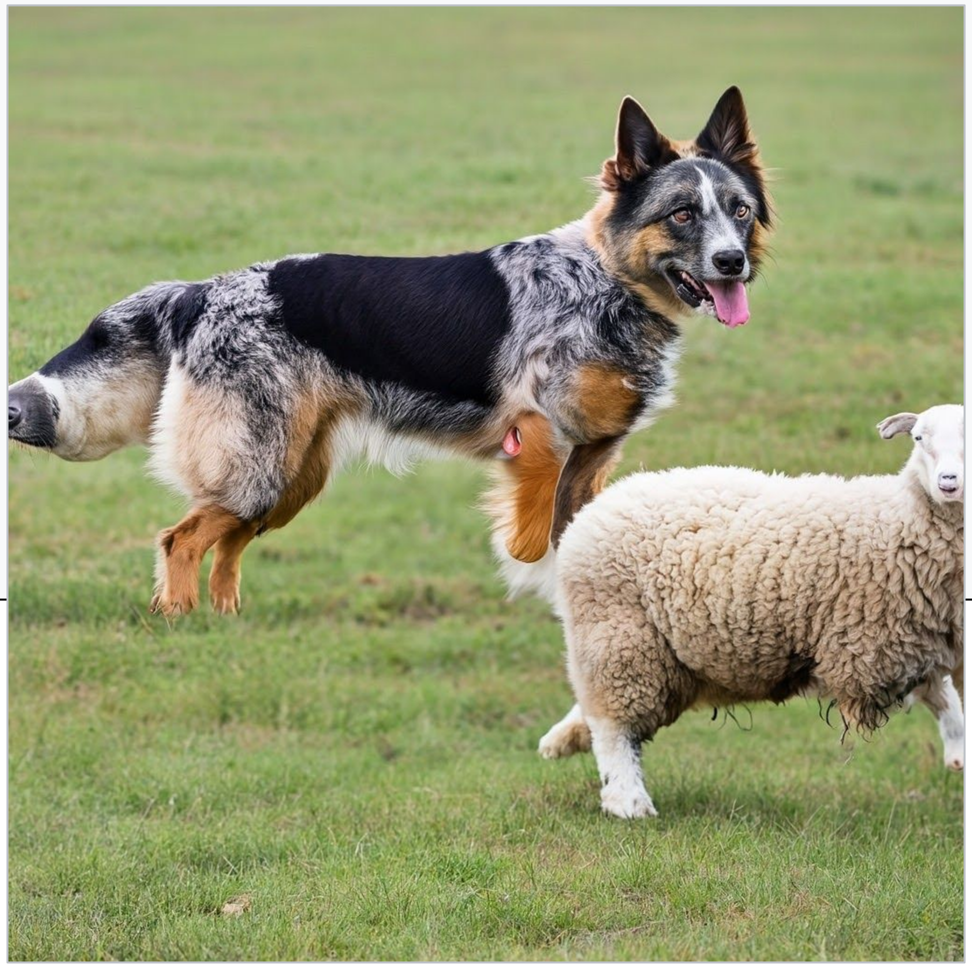
Using Imagen for image generation (in preview)

```
from vertexai.preview.vision_models import ImageQnAModel, 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

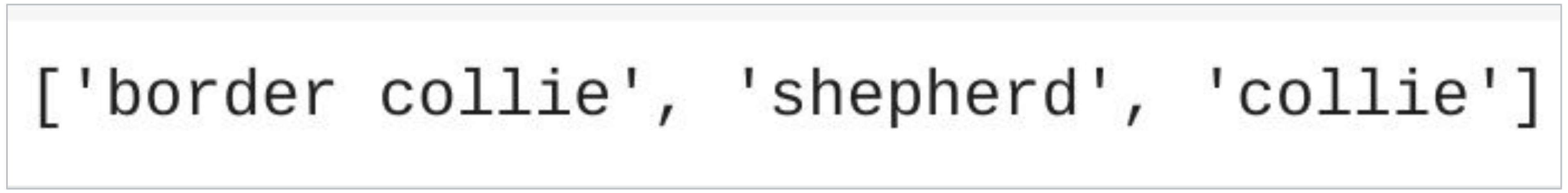
```
from vertexai.vision_models import ImageCaptioningModel, Image

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
```

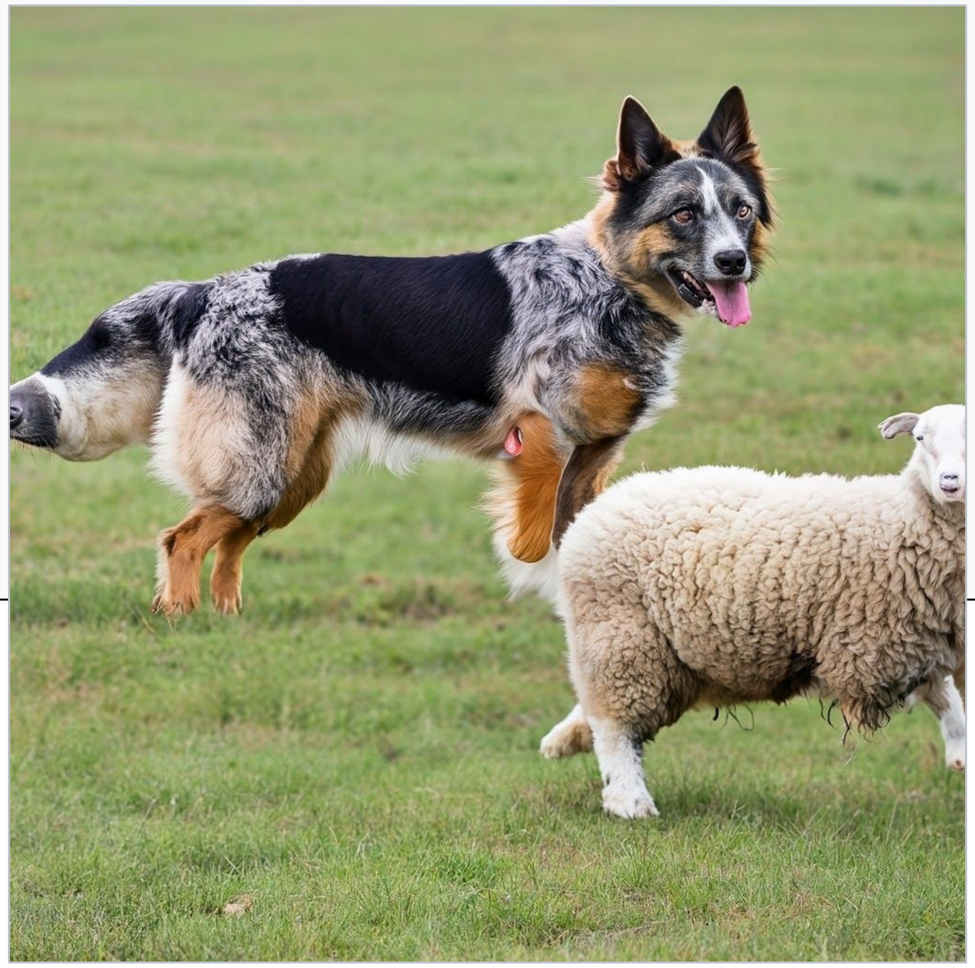


Using Imagen for image Q&A

```
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']
```



Using Gemini for Text Generation

```
import vertexai
from vertexai.preview.generative_models import GenerativeModel, Part

def generate():
  model = GenerativeModel("gemini-pro")
  responses = model.generate_content(
    """Do Border Collies make good pets?""",
    generation_config={
        "max_output_tokens": 2048,
        "temperature": 0.9,
        "top_p": 1}, stream=True,)

  for response in responses:
      print(response.candidates[0].content.parts[0].text)

generate()
```

The API is different when using Gemini. It is not just a matter of specifying a different model.

Using Gemini for Chat

```
def multiturn_generate_content():
    config = {
        "max_output_tokens": 2048,
        "temperature": 0.9,
        "top_p": 1
    }
    model = GenerativeModel("gemini-pro")
    chat = model.start_chat()
    print(chat.send_message("""Hi""", generation_config=config))

multiturn_generate_content()
```

With Gemini, the same model is used for text generation and chat apps.

Using Gemini for Vision

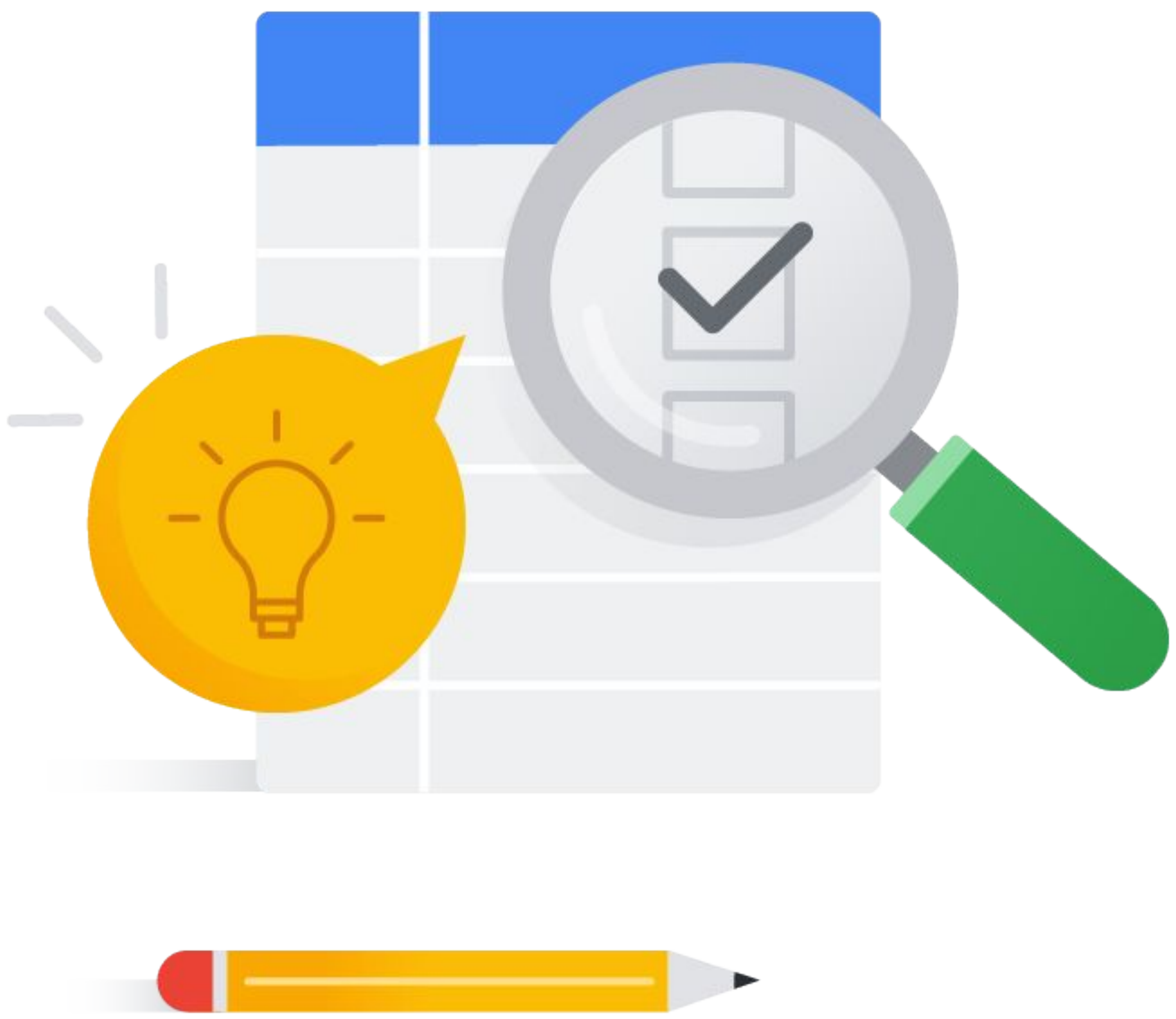
```
from vertexai.preview.generative_models import GenerativeModel, Image

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

Lab

30 min

Lab: Getting Started with the PaLM API for Chatbots



In this module, you learned to ...

01 Program with the PaLM REST API

02 Program Jupyter Notebooks that use the PaLM API

03 Add GenAI 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