ITCS 4101: Introduction to NLP

Coding with Large Language Models using APIs:

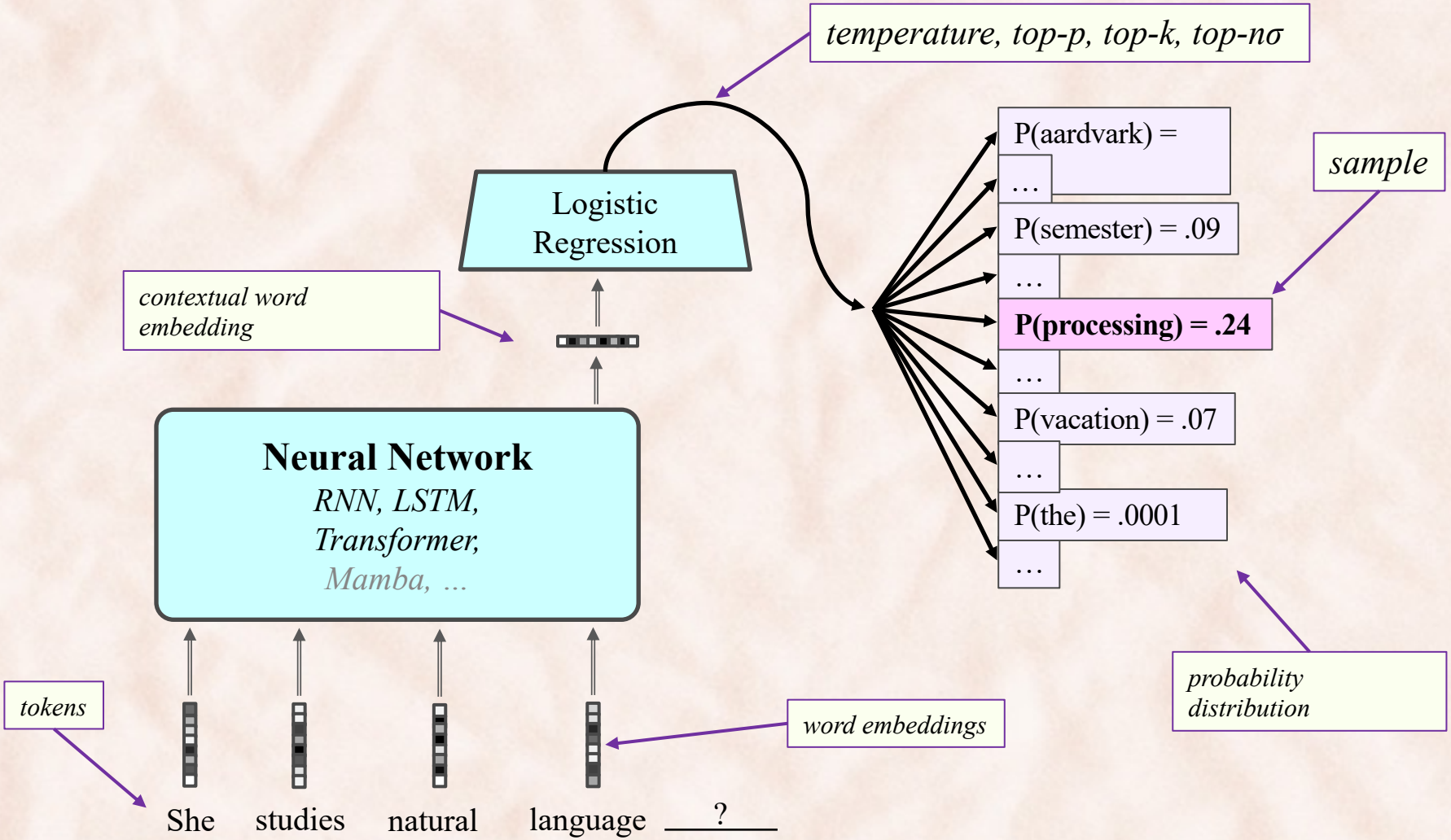
GTP, Gemini, Llama, …

Razvan C. Bunescu

Department of Computer Science @ CCI

rbunescu@uncc.edu

Large Language Models (LLMs)



Application Development Using LLM APIs

We will discuss 3 main options for this class:

1. OpenAI's **GPT** models.
   - Use as a service, pay per token.
   - Open-source versions gpt-oss.
2. Google's **Gemini** models.
   - Free tier, with lower rate limits.
   - Paid tier, with higher rate limits.
   - Need personal Gmail account.
3. Meta's **Llama** models.
   - Free, open-source Llama-3 model, installed on a College server.

Two Ways of Using GPT and Gemini Models

1. Through the **browser app**:
   - ChatGPT at https://chatgpt.com
   - Gemini at https://gemini.google.com/app

2. As a service, through an **API**:
   - **Directly**, in the code:
     - GPT through the Response API.
     - Gemini through the Developer API.
   - **Indirectly**, in the browser:
     - GPT Playground at https://platform.openai.com/chat
     - Google AI Studio at https://aistudio.google.com/prompts/new_chat

OpenAI GPT

Using OpenAI GPT models

- GPT = Generative Pre-trained Transformer.
- **Pre-trained** to "understand" natural language and code:
  - Using a *language modeling* (LM) objective.
- **Fine-tuned** to provide text outputs (answers) in response to their inputs (questions or **prompts**).
  - **Instruction fine-tuning**.
  - **Alignment with human preference** (RLHF with PPO, DPO, …).
- "Programming" with GPT, Gemini, Llama, and other LLMs:
  - Design a "prompt", usually by providing **instructions** and/or some examples of how to successfully complete a task:
    - *zero-shot, few-shot in-context learning, CoT explanations.*

GPT: Setting up the OpenAI API account

- Need to have an OpenAI account:
  - Go to https://platform.openai.com, Log in / Signup.
    - **"Continue with Google", use your UNCC email**.
      - $5 should be enough for the work in this class, see pricing.
      - Go to billing overview, *Set payment* → input credit card, or *Add to credit balance*, input $5.
- Create a secret API key and store it in a .env file:
  - Go to API keys and "+ *create new secret key*".
  - Copy the key and store it in a text file named **.env** as follows
    - OPENAI_API_KEY=…
    - Make sure you save the key, it will not be shown again.
  - Place or copy the **.env** file in the folder you edit and run the notebook.
    - Do not put the secret key in your code!

Required Python Modules

- Install the **openai** module and **python-dotenv** module :
  - pip install openai
  - pip install python-dotenv

```
import os
from openai import OpenAI

from dotenv import load_dotenv, find_dotenv

# Read the local .env file, containing the Open AI secret key.
_ = load_dotenv(find_dotenv())

client = OpenAI(api_key = os.environ['OPENAI_API_KEY'])
```

- Alternatively, use Google Colab instead of JupyterLab:
  - Has modules already installed.
  - But ensure to use Colab's built-in "Secrets" feature to store the keys.

Setting up and reading secret keys in Colab

2. Accessing Colab Secrets (Recommended for sensitive data):

For sensitive information like API keys, it is recommended to use Colab's built-in "Secrets" feature.

- **Set up the Secret:**
  - In your Colab notebook, click the "key" icon on the left sidebar to open the Secrets manager.
  - Add a new secret, providing a name (e.g., `MY_API_KEY`) and its corresponding value.
  - Ensure the "Notebook access" toggle is enabled for your current notebook.
- **Read the Secret in your Notebook:**

Python

```
from google.colab import userdata
import os

# Retrieve the secret value using its name
api_key = userdata.get('MY_API_KEY')

# Optionally, set it as an environment variable for compatibility wit
os.environ['MY_API_KEY'] = api_key

print(f"API Key (first few chars): {api_key[:5]}...") # Print a parti
```

3. Reading from a `.env` file (if you've uploaded one):

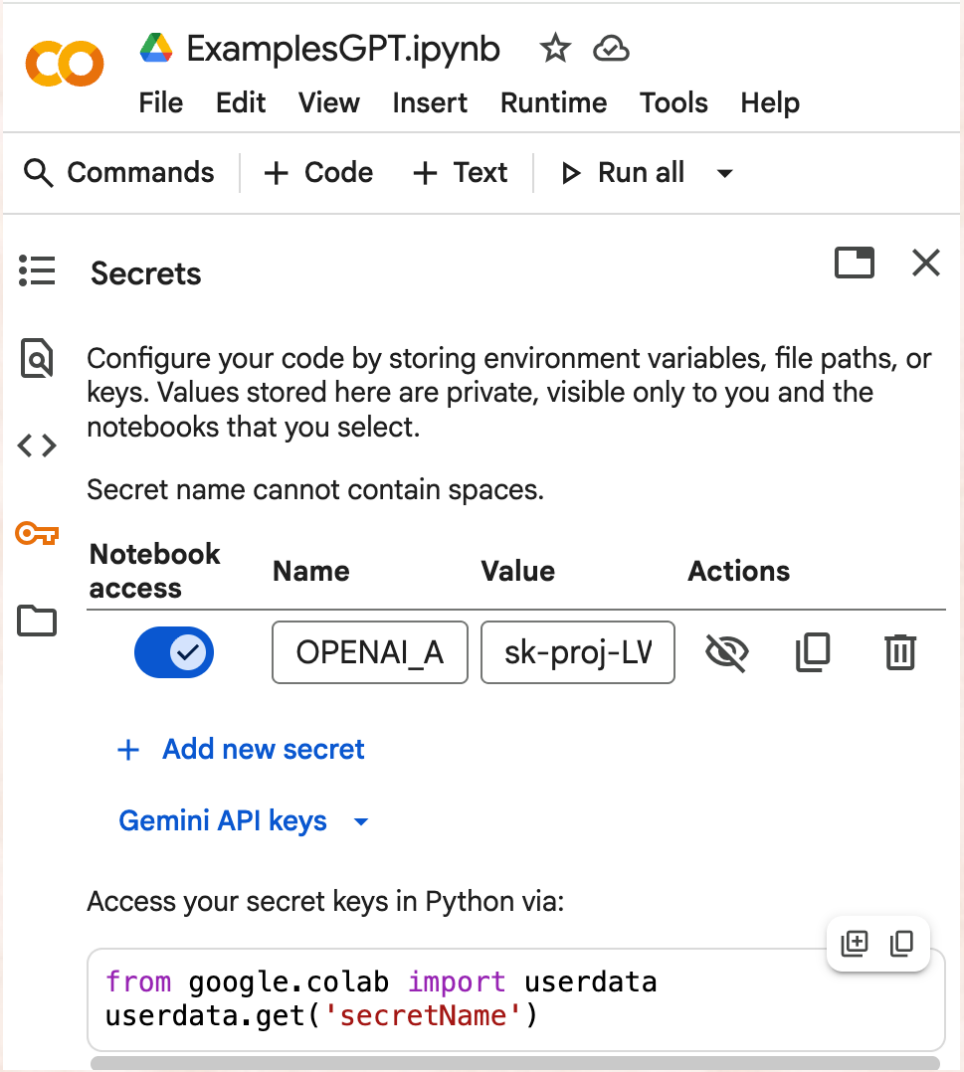
If you've uploaded a `.env` file to your Colab environment, you can use the `python-dotenv` library to load it:

Python

```
!pip install python-dotenv

import dotenv
dotenv.load_dotenv('./.env')

import os
value = os.environ.get('YOUR_VARIABLE_NAME_IN_DOTENV')
print(f"The value from .env is: {value}")
```



Using the Response API

- Take a list of messages as input and return a model-generated message as output.
  - Designed to make multi-turn conversations easy, it's just as useful for single-turn tasks without any conversation.

```
response1 = client.responses.create(
    model = "gpt-5-nano",
    reasoning = {"effort": "low"},
    input = [
        {
            "role": "developer",
            "content": "You are a helpful music historian."
        },
        {
            "role": "user",
            "content": "Who composed The Four Seasons?"
        }
    ]
)

print(response1.output_text)
```

https://platform.openai.com/docs/guides/text

Response API: openai.responses.create

- 3 major roles in the **input** parameter:
  - **Developer**: Optional message, that indicates the LLM persona.
    - Also called a *steering prompt*, sets up the system behavior.
  - **User**: Provides questions, requests, or comments to the assistant.
  - **Assistant**: Previous responses from the LM assistant, or example of desired LM response.
    - **Need to provide the conversation so far, every time** we want to continue with a new user questions.

- Roles interact in a chain of command, with authority levels:

1. **Platform**: Model Spec "platform" sections and system messages
2. **Developer**: Model Spec "developer" sections and developer messages
3. **User**: Model Spec "user" sections and user messages
4. **Guideline**: Model Spec "guideline" sections
5. *No Authority*: assistant and tool messages; quoted/untrusted text and multimodal data in other messages

Response API: openai.responses.create

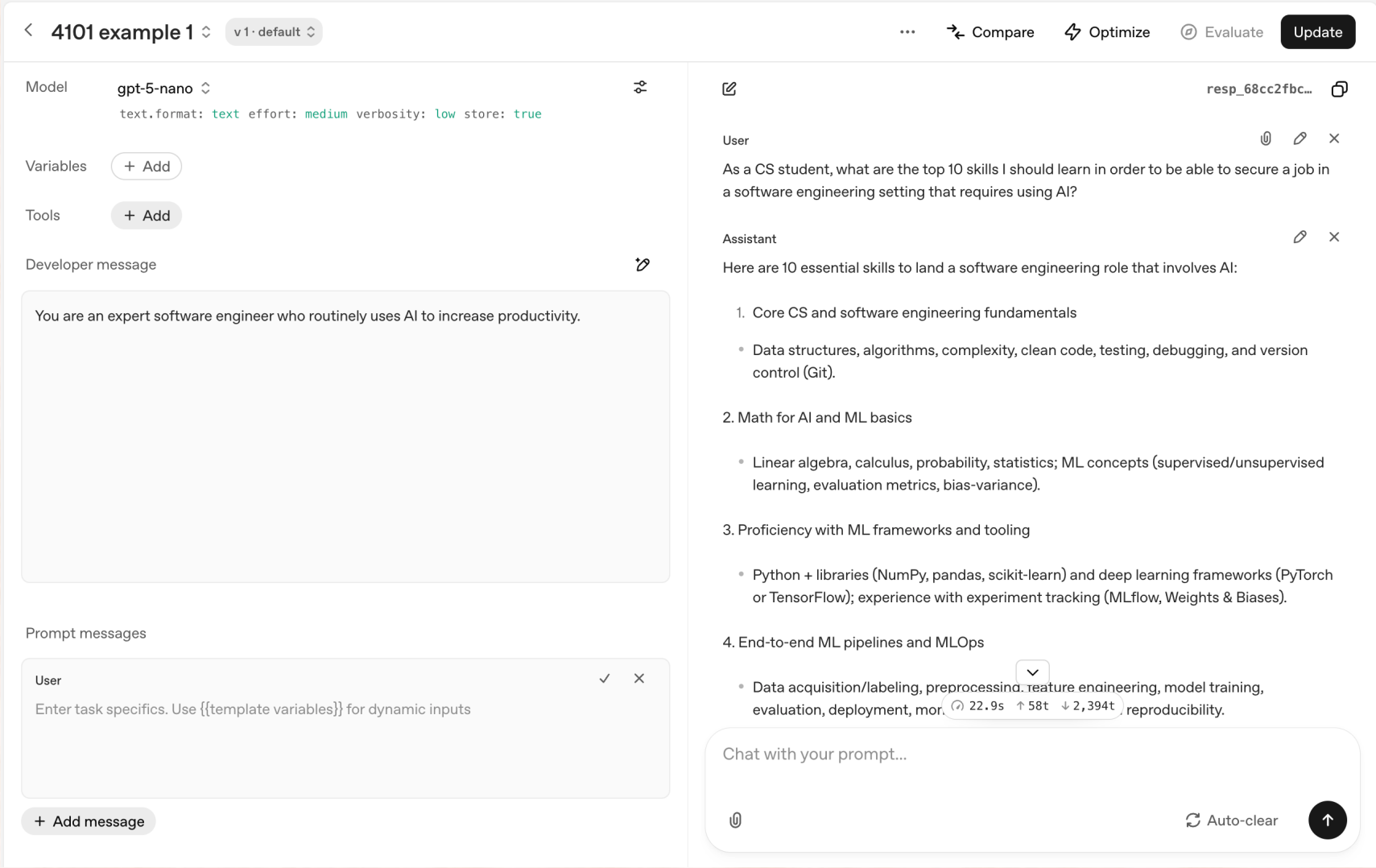
- Other useful parameters:
  - **model**: gpt-5 or gpt-5-min or gpt-5-nano.
  - **max_output_tokens**: maximum # of tokens to generate.
  - **reasoning**: effort level and configuration for reasoning models, default is 'medium'.
  - **tool_choice**: Controls which (if any) tool is called by the model.
  - **temperature**: defaults to 1, but set it to **0 for greedy decoding**.
    - Eliminated in GPT-5!
  - **top_p**: defaults to 1, use 0.1 if you want the LLM to sample tokens only from the top 10% of probability mass, i.e. **nucleus sampling**.
  - presence_penalty, frequence_penalty, logit_bias: penalize or favor repetitions, or certain tokens (later in this course).

https://platform.openai.com/docs/api-reference/responses

Examples with Open AI GPT

- Shown in the Jupyter notebook.

Using the API interactively in GPT Playground



Using the API interactively in GPT Playground

- The Playground facilitates quick stress testing of prompts and parameters, before deploying in code.

- Once the prompt and parameters are ready, you can see the Python or node.js code for the conversation. Copy & Paste into your code.

Google Gemini

Gemini: Setting up the API account

- Need to have a personal Google account:
  - UNCC account will not work (OIT still working on it …).
  - Go to https://ai.google.dev/, Sign in.
  - Continue with **personal account**.
    - See pricing for available models and pricing and rate limits.

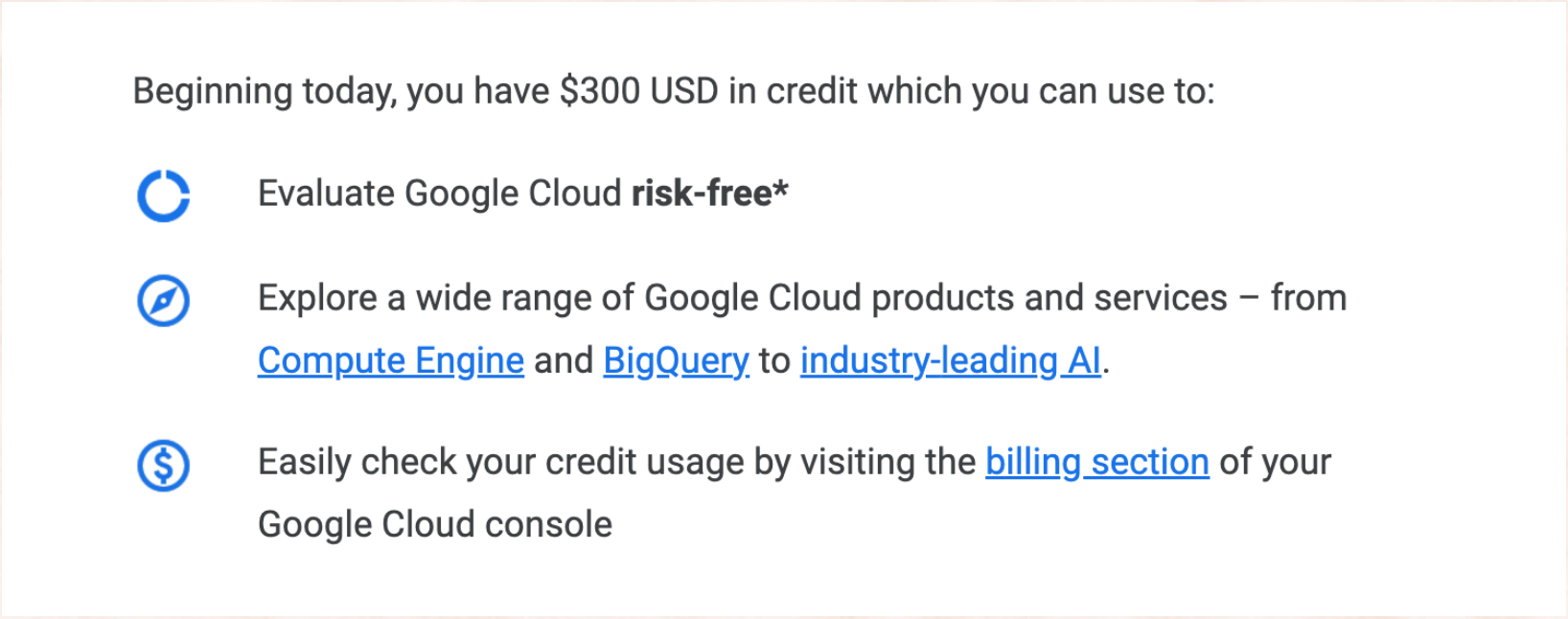
Free Tier | Tier 1 | Tier 2 | Tier 3

Model	RPM	TPM	RPD
Text-out models			
Gemini 2.5 Pro	5	250,000	100
Gemini 2.5 Flash	10	250,000	250
Gemini 2.5 Flash-Lite	15	250,000	1,000
Gemini 2.0 Flash	15	1,000,000	200
Gemini 2.0 Flash-Lite	30	1,000,000	200

- Follow the Gemini API quickstart in Python (next slides).

Gemini: Setting up the API account

- First time API users get $300 free credit for Google Cloud:
  - Including the Gemini API.
  - To be used within 3 months.



Gemini: One-time Setup of API Key

- Create and store a secret API key:
  - Get an API key from Google AI Studio.
    - Click **Create API Key**.
  - Copy the key and store it in a text file named **.env** as follows
    - GEMINI_API_KEY=…

- Place or copy the **.env** file in the folder you edit and run the notebook.
  - Other solutions exist, but this is what we will do in this course.
  - Do not put the secret key in your code!

Required Python Modules

- Install the **google-genai** module:
  - pip install -U google-genai (use pip3)
    - Make sure you have latest version of pip3 and setuptools:
      - pip3 install --upgrade pip
      - python3 -m pip install --upgrade setuptools

- Install the **python-dotenv** module, using one of:
  - pip install python-dotenv

- Alternatively, use Colab instead of Jupyter:
  - Has modules already installed.
  - But ensure to use Colab's built-in "Secrets" feature to store the keys.

Setting up and reading secret keys in Colab

2. Accessing Colab Secrets (Recommended for sensitive data):

For sensitive information like API keys, it is recommended to use Colab's built-in "Secrets" feature.

- **Set up the Secret:**
  - In your Colab notebook, click the "key" icon on the left sidebar to open the Secrets manager.
  - Add a new secret, providing a name (e.g., `MY_API_KEY`) and its corresponding value.
  - Ensure the "Notebook access" toggle is enabled for your current notebook.
- **Read the Secret in your Notebook:**

Python

```
from google.colab import userdata
import os

# Retrieve the secret value using its name
api_key = userdata.get('MY_API_KEY')

# Optionally, set it as an environment variable for compatibility wit
os.environ['MY_API_KEY'] = api_key

print(f"API Key (first few chars): {api_key[:5]}...") # Print a parti
```

3. Reading from a `.env` file (if you've uploaded one):

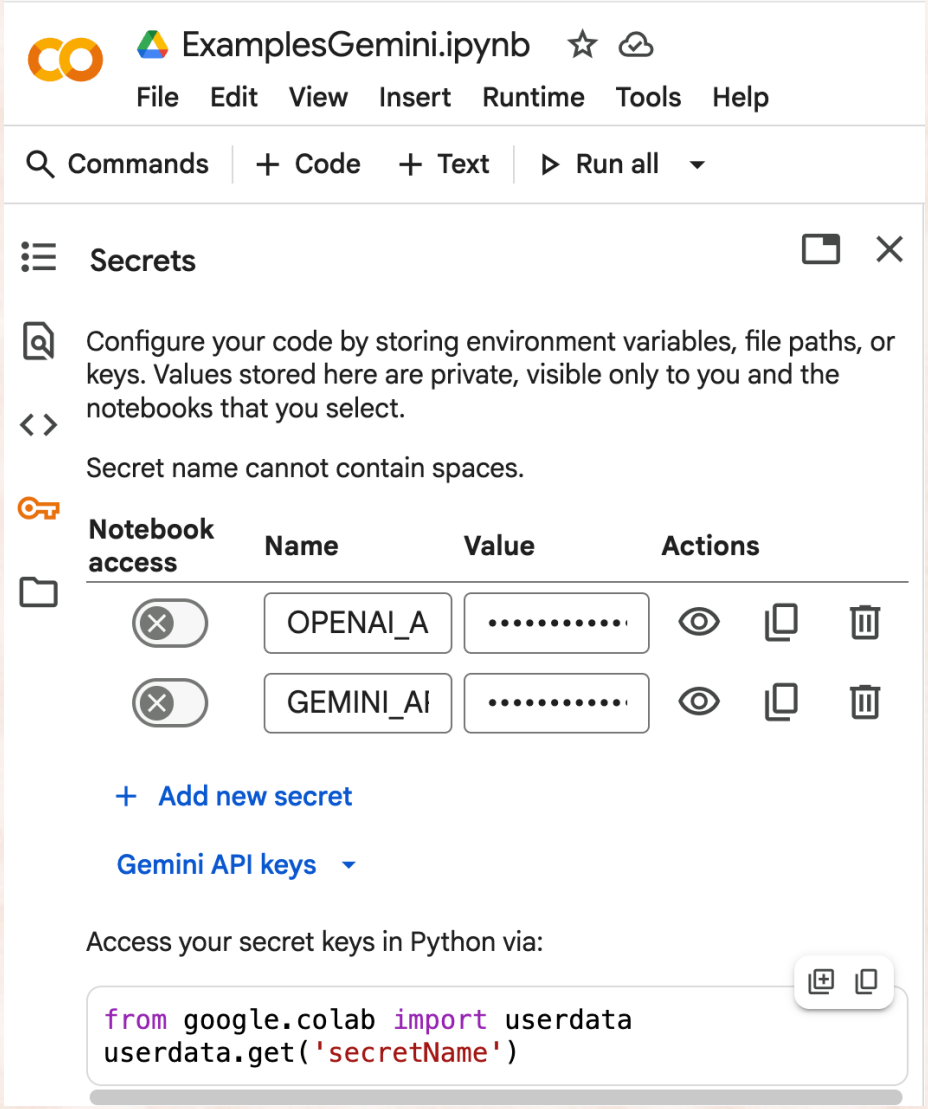
If you've uploaded a `.env` file to your Colab environment, you can use the `python-dotenv` library to load it:

Python

```
!pip install python-dotenv

import dotenv
dotenv.load_dotenv('./.env')

import os
value = os.environ.get('YOUR_VARIABLE_NAME_IN_DOTENV')
print(f"The value from .env is: {value}")
```



Gemini Client Setup and Query

- Create the Client object:

```
import os
from google import genai
from dotenv import load_dotenv, find_dotenv

# Read the local .env file, containing the Gemini secret API key.
_ = load_dotenv(find_dotenv())

client = genai.Client(api_key = os.environ["GEMINI_API_KEY"])
```

- Send a query, using default parameters:

```
query = "Justin sits next to Razvan. One of them is happy and one of them is grumpy. " \
                "The person sitting next to Justin is grumpy. Who is happy?"

response = client.models.generate_content(
    model = "gemini-2.5-flash",
    contents = query
)
print(response.text)
```

Gemini: Multiple Turn Conversations

- Use the **google.genai.chats.Chat** class to manage turns in the conversation:

```
chat = client.chats.create(model = "gemini-2.5-flash")

response = chat.send_message("Who received the Nobel prize in medicine for cancer immunotherapy?")
print(response.text)

response = chat.send_message("Where did they do their research?")
print(response.text)

for message in chat.get_history():
    print(f'role - {message.role}', end = ": ")
    print(message.parts[0].text)
```

★ **Note:** Chat functionality is only implemented as part of the SDKs. Behind the scenes, it still uses the **generateContent** API. For multi-turn conversations, the full conversation history is sent to the model with each follow-up turn.

Hard to find API documentation on how to engage in **multi-turn multimodal conversations** …

When API Documentation Fails, Turn to Google's AI Overview

gemini api generate_content with multiple content fields

AI Mode | All | Videos | Images | Short videos | Forums | Shopping | More | Tools

AI Overview

The Gemini API's `generateContent` method allows for the inclusion of multiple content fields within a single request, particularly when dealing with multimodal inputs or multi-turn conversations.

Multimodal Inputs:

Gemini models are designed to handle various modalities like text, images, and audio. To include multiple content types in a single `generateContent` call, you structure the `contents` field of your request to contain a list of `Part` objects, each representing a different modality.

For example, to send both text and an image:

Code

```
{
  "contents": [
    {
      "role": "user",
      "parts": [
        {
          "text": "Describe this image:"
        },
        {
          "inlineData": {
            "mimeType": "image/jpeg",
            "data": "base64_encoded_image_data"
          }
        }
      ]
    }
  ]
}
```

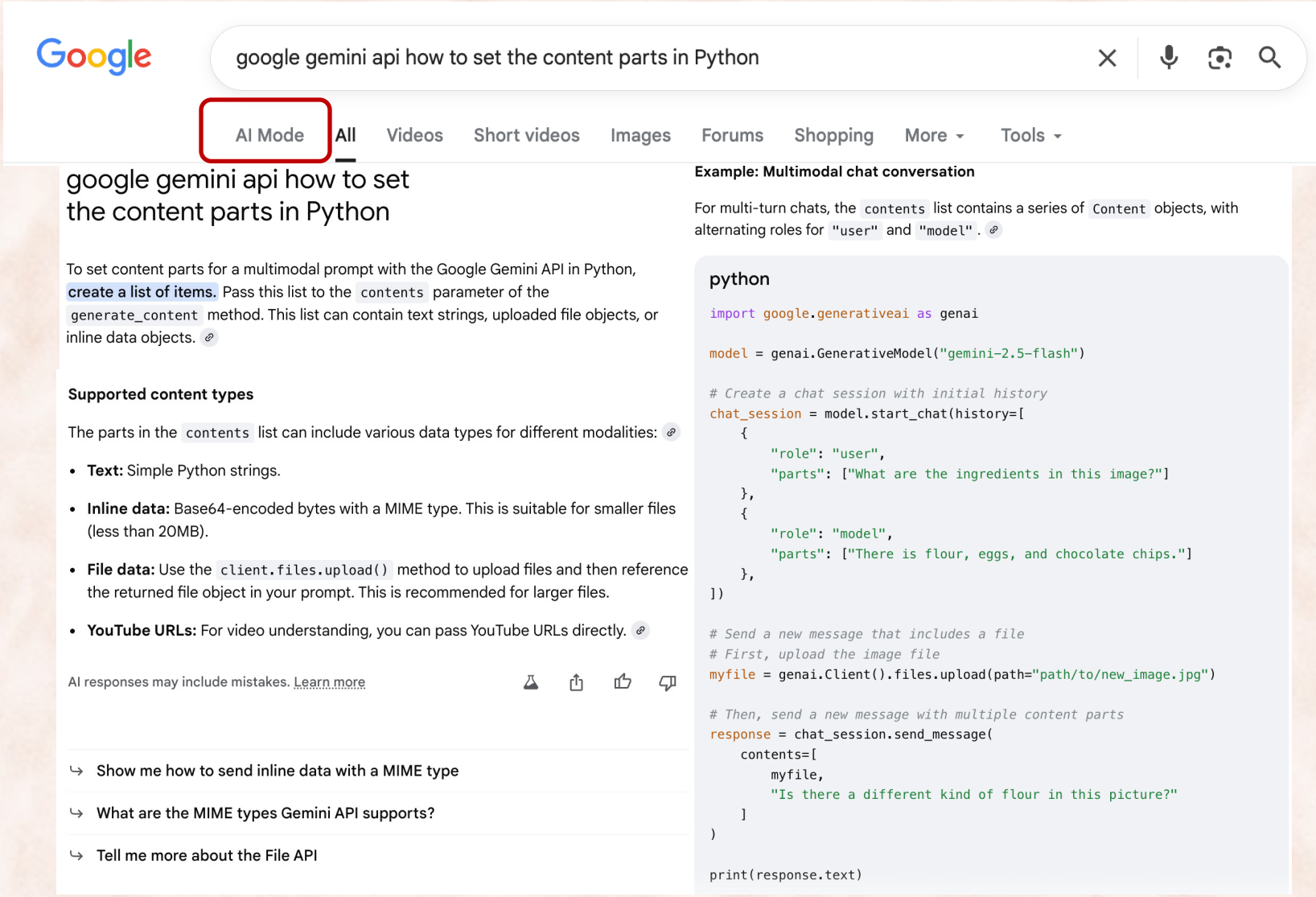
Multi-Turn Conversations (Chat):

When working with chat-like interactions, you can provide the conversation history by including multiple `Content` objects within the `contents` list. Each `Content` object represents a turn in the conversation, specifying the `role` (e.g., "user" or "model") and the `parts` of that turn.

Code

```
{
  "contents": [
    {
      "role": "user",
      "parts": [
        {
          "text": "What is the capital of France?"
        }
      ]
    },
    {
      "role": "model",
      "parts": [
        {
          "text": "The capital of France is Paris."
        }
      ]
    },
    {
      "role": "user",
      "parts": [
        {
          "text": "Tell me more about its history."
        }
      ]
    }
  ]
}
```

Turn to AI Mode



Examples with Google Gemini

- Shown in the Jupyter notebook.

- More examples in the Gemini Developer API documentation:
  - Text generation examples.
  - Image generation examples.
  - Image understanding example.

- Parameters such as temperature, max_output_tokens, … can be set using google.genai.types.GenerateContentConfig().

- More documentation available at:
  - Google Gen AI SDK.
  - Vertex AI.

Meta Llama

Using Llama 3.3-70B Quantized

- **OpenAI.base_url**:
  - An attribute of the OpenAI class.

- **Model name**:
  - Specifies which version of Llama 3 is being utilized.
  - You must be on eduroam to access the model directly. Off campus, you need connect through the educational cluster using VPN.

```
import httpx
from openai import OpenAI

# Set the Llama API base URL.
BASE_URL = "https://cci-llm.charlotte.edu/api/v1"

# Initialize client with SSL verification disabled
client = OpenAI(base_url = BASE_URL,
                http_client = httpx.Client(verify = False),
                api_key = '3jdhdi4xkf-45')

model_name = "Llama-3.3-70B-Instruct"
```

  - Send messages using the original **chat completion API** from OpenAI.

Chat Completion API: *openai.chat.completions.create()*

- Initial API from OpenAI, still has backward support for it.

```
from openai import OpenAI

client = OpenAI(api_key = os.environ['OPENAI_API_KEY'])
```

https://platform.openai.com/docs/api-reference/chat

- Most LLMs support it, including Llama and Gemini.
  - Take a list of messages as input and return a model-generated message as output.
  - Designed to make multi-turn conversations easy, it's just as useful for single-turn tasks without any conversation.

model_name

```
response =  client.chat.completions.create(
    model = ,
    messages = [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Who composed The Four Seasons?"},
        {"role": "assistant", "content": "Antonio Vivaldi composed The Four Seasons."},
        {"role": "user", "content": "For whom were most of his compositions written?"}
    ]
)

print(response.choices[0].message.content)
```

Chat Completion API: *openai.chat.completions.create()*

- 3 major roles in the **messages** parameter:
  - **System**: Optional first message, that indicates the LM persona.
    - Also called a *steering promp*, sets up the system behavior.
    - Default is "You are a helpful assistant".
  - **User**: Provides questions, requests, or comments to the assistant.
  - **Assistant**: Previous responses from the LM assistant, or example of desired LM response.
    - **Need to provide the conversation so far every time** we want to continue with a new user questions.

- Typical input (RE) is **system? user (assistant user)***

Chat Completion API: *openai.chat.completions.create()*

- Other useful parameters:
  - **model**: gpt-5 or gpt-5-min or gpt-5-nano or …
  - **temperature**: defaults to 1, but set it to **0 for greedy decoding**.
    - We'll see how it is implemented when covering *Logistic Regression*.
  - **top_p**: defaults to 1, use 0.1 if you want the LM to sample tokens only from the top 10% of probability mass, i.e. **nucleus sampling**.
  - **n** : defaults to 1, indicates # completions (alternatives) to generate.
  - **max_tokens**: defaults to ∞, maximum # of tokens to generate.
  - presence_penalty, frequence_penalty, logit_bias: penalize or favor repetitions, or certain tokens (later in this course).

Using Llama through the ccAPI

```
# Define the conversation
query = "Justin sits next to Razvan. One of them is happy and one of them is grumpy. " \
                "The person sitting next to Justin is grumpy. Who is happy?"
conversation = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": query}
]

# Send a chat completion request
response =  client.chat.completions.create(
    model = model_name,
    messages = conversation,
    max_tokens = 300,
    temperature = 0
)

reply = response.choices[0].message.content
print(f"Text response: {reply}")
```

Using Llama through the ccAPI

Python & JSON Comprehension

```
question = 'Consider the following monologue from the movie Stalker by Andrei Tarkovsky: ' \
           '"Let them be helpless like children, because weakness is a great thing, and strength is nothing. ' \
           'When a man is just born, he is weak and flexible. When he dies, he is hard and insensitive. ' \
           'When a tree is growing, it\'s tender and pliant. But when it\'s dry and hard, it dies. ' \
           'Hardness and strength are death\'s companions. Pliancy and weakness are expressions of the ' \
           'freshness of being. Because what has hardened will never win." ' \
           'Where else was a similar idea expressed? Provide quotes. Format your answer as a Python dictionary ' \
           'mapping the author or source name to the actual passage expressing a similar idea.'

conversation = [{"role": "system", "content": "You are a helpful librarian."},
            {"role": "user",
             "content": question}]

response = client.chat.completions.create(
    model = model_name,
    messages = conversation,
    max_tokens = 700,
    temperature = 0
)

print(response.choices[0].message.content)
```

Examples with Llama3

- Shown in the Jupyter notebook.

Supplemental Material

- DeepLearning.AI short course on Building with Llama 4.