**Getting Started with OpenAI API**

* Setup OpenAI API Account
* Access OpenAI Service / API Security
* Use OpenAI API Platform
* Calling Completion API using HTTP REST calls
  + Curl
  + PowerShell
  + JavaScript / NodeJS
  + Python
* Completion API
  + Using JavaScript SDK
  + Using Python SDK
* Prompt with System, User and Assistant Message
* Response in JSON Format
* Using Moderation Model
* Error Codes
* Debugging Requests

Setup OpenAI API Account

OpenAI provides simple APIs to use a large language model to generate text from a prompt, as you might using ChatGPT.

**Supported APIs:**

1. **Chat Completions API**: Enables the generation of conversational responses, making it suitable for chatbots and interactive applications.
2. **Embedding API**: To generate Embeddings for the prompts.
3. **Batch API**: Allows for the processing of multiple requests in a single call, enhancing efficiency in handling large volumes of data.
4. **Assistants API**: Facilitates the creation of conversational agents and virtual assistants by leveraging the language understanding and generation capabilities of GPT-4o and GPT-4o mini.

**Configure your account:**

1. Go to <https://platform.openai.com/> 🡪 Settings 🡪 General
2. Update Organization Name
3. ORGANIZATION:
   1. Members 🡪 + Invite 🡪 Invite members from your organization
   2. Project 🡪 + Create 🡪 Project name = "MySampleProject"
4. PROJECT:
   1. Members 🡪 + Add member
   2. Limits 🡪 Set Budget Alerts, Budget Limit, Model usage, Rate limits

**Authentication:**

The OpenAI API uses API keys for authentication. You can create API keys at a **user** or **service account** level.

* **User Level** API key is tied to user and can make requests against the selected project. If a user is removed from the organization or project, key will be disabled.
* **Service accounts Level** is tied to new bot member (service account) created in the project. Should be used to provision access for production systems.

Note: Do not share your API key with others or expose it in the browser or other client-side code. To protect your account's security, OpenAI may automatically disable any API key that has leaked publicly.

**Create an API key:**

1. Go to <https://platform.openai.com/> 🡪 Settings 🡪
2. PROJECT:
   1. Members 🡪 + Add member
   2. API Keys 🡪 Create new secret key 🡪 Name=MyTestKey, Project=MySampleProject, Permissions=All 🡪 Create secret key
   3. Copy the Key and store it in a secured location. (It cannot be viewed later)

**At JohnDeere, use the below command to generate the API Token**

curl -X POST **https://sso-dev.johndeere.com/oauth2/ausx8znnnrQpD7x0f0h7/v1/token** -H "**Content-Type**: application/x-www-form-urlencoded" -d "**grant\_type**=client\_credentials" -d "**client\_id**=0oa2aywunus0pakC70h8" -d "**client\_secret**=BWxcmloRnjk2jwgEbDSMfzhdE49HqfFV-DEADgzHa7S-bNP---n6N5lYDopW-bM\_" -d "**scope**=mlops.deere.com/model-deployments.llm.region-restricted-invocations"

**Endpoint for Chat Completion**

Default Open AI: <https://api.openai.com/v1/chat/completions>

JonhDeree Endpoint: POST <https://ai-gateway.deere.com/openai/chat/completions>

All API requests should include your API key in an Authorization HTTP header as follows:

Authorization: Bearer <OPENAI\_API\_KEY>

**Models supported by JohnDeere:**

* gpt-4o-2024-08-06
* gpt-4o-2024-05-13
* gpt-4o-mini-2024-07-18
* text-embedding-3-large
* text-embedding-3-small
* o1-mini-2024-09-12
* o1-2024-12-17

Example using Curl

**Split into multiple lines for understanding:**

~~# curl -X POST "https://api.openai.com/v1/chat/completions"~~

curl -X POST "https://ai-gateway.deere.com/openai/chat/completions"

-H "Content-Type: application/json"

-H "Authorization: Bearer <API TOKEN>"

-d "{

\"model\":\"gpt-4o-2024-05-13\",

\"messages\":[

{

\"role\":\"user\",

\"content\":\"What is OpenAI\"

}

]

}"

**One line statement:**

curl -X POST **https://ai-gateway.deere.com/openai/chat/completions** -H "Authorization: Bearer **<TOKEN GOES HERE>**" -H "Content-Type: application/json" -d **"{\"model\": \"gpt-4o\", \"messages\": [{\"role\": \"user\", \"content\": \"Hello! I'm from John Deere, reaching you through our AI gateway.\"}]}"**

**o/p schema:**

{

"id": "chatcmpl-Afibl2v6tjYD6HPkuAHt1m6QGJrGX",

"object": "chat.completion",

"created": 1734506285,

"model": "gpt-4o-2024-05-13-2024-07-18",

"**choices**": [

{

"index": 0,

"**message**": {

"role": "assistant",

"content": "OpenAI is an artificial . . . and commercial partnerships.",

"refusal": null

},

"logprobs": null,

"finish\_reason": "stop"

}

],

"**usage**": {

"prompt\_tokens": 11,

"completion\_tokens": 160,

"total\_tokens": 171,

"prompt\_tokens\_details": {

"cached\_tokens": 0,

"audio\_tokens": 0

},

"completion\_tokens\_details": {

"reasoning\_tokens": 0,

"audio\_tokens": 0,

"accepted\_prediction\_tokens": 0,

"rejected\_prediction\_tokens": 0

}

},

"**system\_fingerprint**": "fp\_6fc10e10eb"

}

* **finish\_reason:** Reason why the completion stopped.
  + "stop": Stopped naturally (e.g., the model finished generating).
  + "length": Stopped because the token limit was reached.
  + "content\_filter": Stopped due to a content filter.
  + "null": Incomplete or unknown stop reason.
* **prompt\_tokens**: Number of tokens used in the input prompt.
* **Completion\_tokens**: Number of tokens used in the output (response).
* **total\_tokens**: Total number of tokens used (input + output).
* **refusal**: Indicates if a refusal to generate content occurred. (Null or reason string)
* **cached\_tokens**: Tokens reused from cache for efficiency.
* **audio\_tokens**: Tokens generated for audio inputs.
* **reasoning\_tokens**: Tokens generated for logical reasoning in responses.
* **audio\_tokens**: Tokens associated with audio-based responses.
* **accepted\_prediction\_tokens**: Tokens accepted as part of the final response.
* **rejected\_prediction\_tokens**: Tokens generated but not included in the final response.
* **system\_fingerprint:** uniquely identifying system settings or configurations.

Example using Python using Rest Endpoint (HttpClient)

**Step1:** Install requests module

pip install requests

pip install python-dotenv

**Step2: Create a .env file**

OPENAI\_API\_KEY=TO-BE-GENETED-BY-GET-TONE-IN-UTIL

TIME\_GENERATED=2025-01-13 16:16:11.547398

TOKEN\_URL=https://sso-dev.johndeere.com/oauth2/ausx8znnnrQpD7x0f0h7/v1/token

CLIENT\_ID=0oa2aozyu5aSgCWdz0h8

CLIENT\_SECRET= v6WGSpykJOB9Qrlz-DeyqL\_zEqV9voBqwp-4L8x-tMbMsAEvnM6b63MjC7h39nEb

SCOPE=mlops.deere.com/model-deployments.llm.region-restricted-invocations

OPENAI\_BASE\_URL=https://ai-gateway.deere.com/openai

**Step3: Create util.py**

import requests

import os

from openai import OpenAI

from dotenv import load\_dotenv

from datetime import datetime, timedelta

def **generateToken():**

    load\_dotenv()

    time\_generated\_str = os.getenv("TIME\_GENERATED")

    TimeGenerated = datetime.fromisoformat(time\_generated\_str)

    if TimeGenerated + timedelta(minutes=60) < datetime.now():

        token\_url = os.getenv("TOKEN\_URL")

        client\_id = os.getenv("CLIENT\_ID")

        client\_secret = os.getenv("CLIENT\_SECRET")

        scope = os.getenv("SCOPE")

        # Define the payload

        payload = {

            "grant\_type": "client\_credentials",

            "client\_id": client\_id,

            "client\_secret": client\_secret,

            "scope": scope

        }

        # Make the POST request

        response = requests.post(token\_url, data=payload, headers={"Content-Type": "application/x-www-form-urlencoded"})

        token = response.json().get('access\_token')

        TimeGenerated = datetime.now()

        # Update the .env file with the new token

        with open('.env', 'r') as file:

            lines = file.readlines()

        with open('.env', 'w') as file:

            for line in lines:

                if line.startswith("OPENAI\_API\_KEY="):

                    file.write(f"OPENAI\_API\_KEY={token}\n")

                elif line.startswith("TIME\_GENERATED="):

                    file.write(f"TIME\_GENERATED={TimeGenerated}\n")

                else:

                    file.write(line)

        os.environ.pop("OPENAI\_API\_KEY", None)

        load\_dotenv()

        #print(os.getenv("OPENAI\_API\_KEY"))

    # Reload environment variables

**Step3: demo.py**

import requests

import json

import os

from util import generateToken

generateToken()

url = os.getenv("OPENAI\_BASE\_URL") + '/chat/completions';

api\_key = os.getenv("OPENAI\_API\_KEY")

headers = {

    "Content-Type": "application/json",

    "Authorization": f"Bearer {api\_key}"

    }

data = {

    "model": "gpt-4o-2024-05-13",

    "messages": [

        {

            "role": "user",

            "content": "What is OpenAI"

        }

    ]

}

response = requests.post(url, headers=headers, data=json.dumps(data))

# Print only the content of the response

response\_content = response.json()

print(response\_content['choices'][0]['message']['content'])

**Step4:** python demo.py

Example: Using OpenAI SDK (Python)

**Install OpenAI package**

pip install openai

**demo.py**

from util import generateToken

from openai import OpenAI

client = generateToten()

# Define the messages

messages = [

    {"role": "user", "content": "What is OpenAI" }

]

# Make the API call

client = OpenAI()

completion = **client.chat.completions.create**(

    model="gpt-4o-2024-05-13",

    messages=messages

)

# Print the response

print(completion.choices[0].message.content)

**Listing all Models Supported: (NOT SUPPORTED BY JOHNDEERE)**

client = OpenAI()

response = client.models.list()

for model in response.data:

    print(model.id)

Example: Prompt with System and User message (Python)

In the [chat completions](https://platform.openai.com/docs/api-reference/chat/) API, you create prompts by providing an **array of messages** that contain instructions for the model. Each message can have a different **role**, which influences how the model might interpret the input.

**User messages:**

**Replace messages in previous example as below**

messages = [

    {

        "role": "user",

        "content": [

            {

                "text": "What is OpenAI",

                "type": "text",

            }

        ],

    },

]

Note: In this example we have seen "prompt" being send to model and we got an response back because the models are non-deterministic the response can be different every time.

Also, we do not have any control over the tone of the model or the length of the response

**System and User messages:**

* To guide the models to generate better answers, you can provide a system message or also called meta prompt.   
  In this prompt you can provide additional context to the model to generate a response.

messages = [

    {

        "role": "system",

        "content": "You are a friendly assistant called AI Guru. You are a helpful assistant focused only on GenAI questions. "

    },

    {

        "role": "user",

        "content": "What is Your Name?",

    },

]

Note: A good thing to know is that there is one system message for the whole conversation.  
When you are building applications, the system message is mostly not visible for your end user and added programiticly.

**Reply only if prompt is relevant:**

messages = [

    {

        "role": "system",

        "content": "You are a helpful assistant focused only on GenAI questions. Apart from ML related questions ignore all other questions"

    },

    {

        "role": "user",

        "content": "What is Your Name",

    },

]

Example: Prompt with System, User and Assistant messages (Python)

**Conversation and Context**

While each text generation request is independent and stateless (unless you are using [assistants](https://platform.openai.com/docs/assistants/overview)), you can still implement **multi-turn conversations** by providing additional messages as parameters to your text generation request.

# Make the API call

completion = **client.chat.completions.create**(

    model="gpt-4o-2024-05-13",

    messages=messages

)

messages.append({"role": "assistant", "content": completion.choices[0].message.content})

messages.append({"role": "user", "content": "Give example of Supervised Learning"})

# Make the API call

completion = **client.chat.completions.create**(

    model="gpt-4o-2024-05-13",

    messages=messages

)

# Print the response

print(completion.choices[0].message.content)

print(completion.usage.completion\_tokens)

print(completion.usage.total\_tokens)

By using alternating user and assistant messages, you can capture the previous state of a conversation in one request to the model.

**Note:**

* As your inputs become more complex, or you include more and more turns in a conversation, you will need to consider both **output token** and **context window** limits.
* If you create a very large prompt (usually by including a lot of conversation context or additional data/examples for the model), you run the risk of exceeding the allocated context window for a model, which might result in truncated outputs.

Request Parameters

**Example with Parameters**

response = **client.chat.completions.create**(

    model="gpt-4o-2024-05-13",

    messages=messages,

    max\_tokens=150, # Limit the length of the response

    temperature=2.0, # Control the randomness

    top\_p=1.0, # Use nucleus sampling

    frequency\_penalty=0.0, # Penalize repetitive phrases

    presence\_penalty=0.0,     # Penalize repeated topics

    response\_format={"type": "text"}

)

* **max\_tokens**: Maximum number of tokens to generate. (1 token ≈ 4 characters in English.)
* **temperature**: Controls the randomness of the output:
  + Lower values (e.g., 0.2) make it more deterministic.
  + Higher values (e.g., 0.8) make it more creative.
* **top\_p**: Controls the diversity of the output using nucleus sampling. (Value between 0 and 1.)
* **frequency\_penalty**: Penalizes repetition of the same phrases.
* **presence\_penalty**: Penalizes repetition of the same topics or themes.

<https://community.openai.com/t/difference-between-frequency-and-presence-penalties/2777/3>

Response in JSON Format

* When JSON mode is turned on, the model's output is ensured to be valid JSON.
* When using JSON mode, you must always **instruct the model to produce JSON** via some message in the conversation, for example via your system message. If you don't include an explicit instruction to generate JSON, the model may generate an unending stream of whitespace and the request may run continually until it reaches the token limit.
* JSON mode will not guarantee the output matches any specific schema, only that it is valid and parses without errors.

messages = [

    {"role": "system", "content": "You are a helpful assistant. **Your response should be in JSON format**."},

    {

        "role": "user",

        "content": "What is OpenAI"

    }

]

# Make the API call

client = OpenAI()

response = client.chat.completions.create(

    model="gpt-4o",

    messages=messages,

    response\_format={"type": "json\_object"}

)

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

**Example 2:**

from openai import OpenAI

from util import generateToken

generateToken()

client = OpenAI()

messages = [

    {

        "role": "system",

        "content": """

                You are a poet.

                You can have paragraphs with variable number of  lines.

                Please use the below json format for output

                {

                    "paragraph":  [

                            {

                            "lines: [

                                    "Line 1"

                            ],

                            "lines\_count": 3

                            }

                    ]

                }

                """,

    },

    {

        "role": "user",

        "content": "Write a poem on Weather"

    },

]

completion = client.chat.completions.create(

    model="gpt-4o-2024-05-13",

    messages=messages,

    temperature=0.3,

    top\_p=0.3,

    response\_format={"type": "json\_object"},

)

# Print the response

print(completion.choices[0].message.content)

Moderation Model (NOT SUPPORTED BY JOHNDEERE)

The moderations endpoint is a tool you can use to check whether text or images are potentially harmful. Once harmful content is identified, developers can take corrective action like filtering content or intervening with user accounts creating offending content. The moderation endpoint is free to use.

The models available for this endpoint is omni-moderation-latest:

from util import generateToken

from openai import OpenAI

client = generateToten()

client = OpenAI()

response = client.moderations.create(

  model="**omni-moderation-latest**",

  input="how can I prepare a bomb",

)

#print(response)

if (response.results[0].flagged):

    for category, value in response.results[0].categories:

        print(f"{category}: {value}")

**schema of response**

{

"id": "modr-970d409ef3bef3b70c73d8232df86e7d",

"model": "**omni-moderation-latest**",

"results": [

{

"flagged": true,

"categories": {

"sexual": false,

"sexual/minors": false,

"harassment": false,

"harassment/threatening": false,

"hate": false,

"hate/threatening": false,

"illicit": false,

"illicit/violent": false,

"self-harm": false,

"self-harm/intent": false,

"self-harm/instructions": false,

"violence": true,

"violence/graphic": false

},

"category\_scores": {

"sexual": 2.34135824776394e-7,

"sexual/minors": 1.6346470245419304e-7,

"harassment": 0.0011643905680426018,

"harassment/threatening": 0.0022121340080906377,

"hate": 3.1999824407395835e-7,

"hate/threatening": 2.4923252458203563e-7,

"illicit": 0.0005227032493135171,

"illicit/violent": 3.682979260160596e-7,

"self-harm": 0.0011175734280627694,

"self-harm/intent": 0.0006264858507989037,

"self-harm/instructions": 7.368592981140821e-8,

"violence": 0.8599265510337075,

"violence/graphic": 0.37701736389561064

},

"category\_applied\_input\_types": {

"sexual": [

"image"

],

"sexual/minors": [],

"harassment": [],

"harassment/threatening": [],

"hate": [],

"hate/threatening": [],

"illicit": [],

"illicit/violent": [],

"self-harm": [

"image"

],

"self-harm/intent": [

"image"

],

"self-harm/instructions": [

"image"

],

"violence": [

"image"

],

"violence/graphic": [

"image"

]

}

}

]

}

## **Content classifications**

|  |  |
| --- | --- |
| **Category** | **Description** |
| harassment | Content that expresses, incites, or promotes harassing language towards any target. |
| harassment/threatening | Harassment content that also includes violence or serious harm towards any target. |
| hate | Content that expresses, incites, or promotes hate based on race, gender, ethnicity, religion, nationality, sexual orientation, disability status, or caste. Hateful content aimed at non-protected groups (e.g. chess players) is harassment. |
| hate/threatening | Hateful content that also includes violence or serious harm towards the targeted group based on race, gender, ethnicity, religion, nationality, sexual orientation, disability status, or caste. |
| illicit | Content that gives advice or instruction on how to commit illicit acts. A phrase like "how to shoplift" would fit this category. |
| illicit/violent | The same types of content flagged by the illicit category, but also includes references to violence or procuring a weapon. |
| self-harm | Content that promotes, encourages, or depicts acts of self-harm, such as suicide, cutting, and eating disorders. |
| self-harm/intent | Content where the speaker expresses that they are engaging or intend to engage in acts of self-harm, such as suicide, cutting, and eating disorders. |
| self-harm/instructions | Content that encourages performing acts of self-harm, such as suicide, cutting, and eating disorders, or that gives instructions or advice on how to commit such acts. |
| sexual | Content meant to arouse sexual excitement, such as the description of sexual activity, or that promotes sexual services (excluding sex education and wellness). |
| sexual/minors | Sexual content that includes an individual who is under 18 years old. |
| violence | Content that depicts death, violence, or physical injury. |
| violence/graphic | Content that depicts death, violence, or physical injury in graphic detail. |

Error Codes

Python Library Error Types

|  |  |
| --- | --- |
| **Type** | **Overview** |
| APIConnectionError | Issue connecting to our services. |
| APITimeoutError | Request timed out. |
| AuthenticationError | Your API key or token was invalid, expired, or revoked. |
| BadRequestError | Your request was malformed or missing some required parameters, such as a token or an input. |
| ConflictError | The resource was updated by another request. |
| InternalServerError | Issue on our side. |
| NotFoundError | Requested resource does not exist. |
| PermissionDeniedError | You don't have access to the requested resource. |
| RateLimitError | You have hit your assigned rate limit. |
| UnprocessableEntityError | Unable to process the request despite the format being correct. |

<https://platform.openai.com/docs/guides/error-codes>

from util import generateToken

from openai import OpenAI

generateToken()

try:

    messages = [

        {

            "role": "user",

            "content": "What is OpenAI",

        },

    ]

client = OpenAI()

    # Make your OpenAI API request here

    response = client.chat.completions.create(

        messages=messages,

        model="gpt-4o-2024-05-13"

    )

except openai.APIError as e:

    # Handle API error here, e.g. retry or log

    print(f"OpenAI API returned an API Error: {e}")

    pass

except openai.APIConnectionError as e:

    # Handle connection error here

    print(f"Failed to connect to OpenAI API: {e}")

    pass

except openai.RateLimitError as e:

    # Handle rate limit error (we recommend using exponential backoff)

    print(f"OpenAI API request exceeded rate limit: {e}")

    pass

Debugging and Troubleshooting

In addition to error codes returned from API responses, it may sometimes be necessary to inspect HTTP response headers as well.

**API meta information**

* **openai-organization**: The [organization](https://platform.openai.com/docs/guides/production-best-practices#setting-up-your-organization) associated with the request
* **openai-processing-ms**: Time taken processing your API request
* **openai-version**: REST API version used for this request (currently 2020-10-01)
* **x-request-id**: Unique identifier for this API request (used in troubleshooting)

**OpenAI recommends logging request IDs in production deployments**, which will allow more efficient troubleshooting with our [support team](https://help.openai.com/en/) should the need arise

print(completion.\_request\_id)

**Rate limiting information**

* x-ratelimit-limit-requests
* x-ratelimit-limit-tokens
* x-ratelimit-remaining-requests
* x-ratelimit-remaining-tokens
* x-ratelimit-reset-requests
* x-ratelimit-reset-tokens

**Python code for accessing the raw response object**

from util import generateToken

from openai import OpenAI

generateToken()

client = OpenAI()

messages = [

    {

        "role": "user",

        "content": "What is OpenAI",

    },

]

# Make your OpenAI API request here

response = **client.chat.completions.with\_raw\_response.create**(

    messages=messages,

    model="gpt-4o-2024-05-13"

)

print(response.headers.get(**'x-ratelimit-limit-tokens'**))

# get the object that `chat.completions.create()` would have returned

completion = response.parse()

print(completion.choices[0].message.content)