



Cal Poly Pomona

OpenAI-API Interaction

ECE 4317 - Intelligence Systems for Engineers

Group 4

By Daniele Ricciardelli & Nathaniel Case

April 24, 2025

1. Abstract.....	3
2. Introduction.....	3
3. OpenAI API.....	4
4. Python Scripting.....	6
4.1 import openai.....	6
4.2 from PIL import Image.....	7
4.3 import pytesseract.....	7
4.4 import pygetwindow as gw.....	7
4.5 import pyautogui.....	8
4.6 import tkinter as tk.....	8
4.6 OpenCV.....	9
5. First results.....	9
6. Automation.....	10
6.1 OpenCV.....	10
6.2 Implementation.....	10
6.3 Code.....	10
7. Plugin thoughts.....	11
8. Conclusion.....	12

1. Abstract

This project explores the practical integration of the *ChatGPT API* into an automated Python tool designed to assist with quiz-style online assignments. Initially conceived as a way to bypass the repetitive task of copy-pasting or using the snipping tool to input questions into ChatGPT, the tool evolved significantly during development. At its core, the system captures screen content, extracts text using optical character recognition (OCR), and feeds that information into ChatGPT to return concise answers, all without user intervention.

As testing progressed, the potential of this automation became increasingly evident. By incorporating automated clicking mechanisms, the project moved beyond simply reducing cognitive effort to eliminating the need for manual interaction altogether. This shift revealed broader applications for the tool, including dynamic task execution and real-time interaction with on-screen content.

The design merges multiple libraries, *PyAutoGUI* and *PyTesseract* for visual input, *OpenAI* for the AI interface, *PyGetWindow* to select a tab (target), and *Tkinter* for user interaction. When all these are combined, it grants the program both “eyes” and a “brain.” What began as a simple convenience grew into a versatile prototype with potential use cases ranging from academic support tools to intelligent interface plugins.

2. Introduction

The inspiration for this project began with a simple question: How easy would it be to complete an online assignment using GPT alone? In many general education (GE) courses and web-based assessments, questions are typically presented in multiple-choice format. Despite their simplicity, these platforms often fall short in user experience, they don’t always reveal the correct answers or offer any explanation when a mistake is made.

Driven by curiosity, I began experimenting with ChatGPT to assist in solving these types of assignments. The results were promising. GPT consistently offered useful guidance and accurate answers if provided the necessary context. However, this process quickly proved repetitive: capturing screenshots, switching tabs, uploading images, typing prompts, waiting for responses, and then navigating back to submit the answer. The idea of reducing this tedious back-and-forth became the project's foundation.

From this inefficiency emerged the goal: to automate the entire interaction pipeline. What if we could remove the manual effort entirely and let the system handle everything, from recognizing the question to submitting the response? This project was born from that challenge, aiming to streamline and enhance how users interact with AI in real-time educational contexts.

3. OpenAI API

To effectively automate the question-solving process, we needed access to a powerful large language model (LLM). Naturally, we turned to what has become the go-to solution for many in the U.S., OpenAI's ChatGPT, specifically the GPT-4o model. Its combination of high accuracy, adaptability, and familiarity made it the ideal choice for this project.

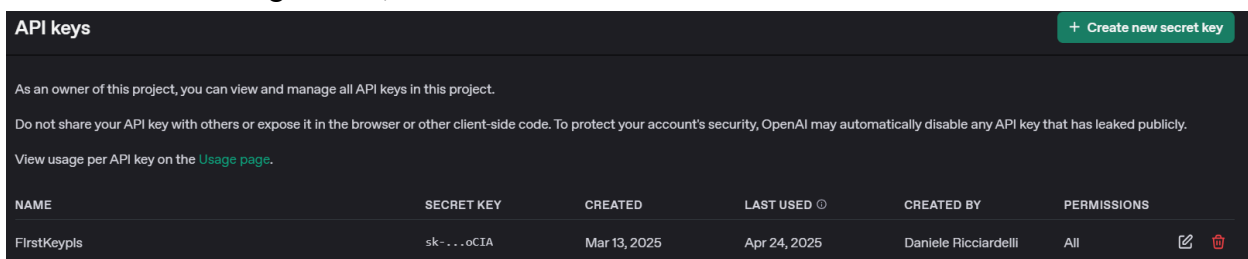
The first step was gaining access to OpenAI's API. We registered through the official platform at <https://platform.openai.com> and purchased \$10 worth of API credits. Notably, OpenAI credits do expire after 12 months if unused, but this window provides ample time for experimentation and development. While the minimum purchase for most users is \$5, we opted for \$10 as a safe buffer for extended testing and scalability.

To our surprise, the tool proved to be extremely efficient in its use of credits. After 146 individual API requests, we had only used \$0.26 of the \$10 balance. This efficiency was achieved by constraining the maximum number of tokens per response - typically set between 500 to 1000 tokens, depending on the testing phase. These settings ensured fast, focused responses while maintaining a negligible cost.

This accessibility and low operating cost made OpenAI's API not just an intelligent choice, but a sustainable one, paving the way for real-time AI-powered automation without breaking the bank.

The most important parts of the API are these two: API keys & Usage. Realistically, with access to only these 2 anybody can easily manipulate the models used and monitor the usage of these.

When creating an API, it will look like this:



NAME	SECRET KEY	CREATED	LAST USED	CREATED BY	PERMISSIONS
FirstKeypls	sk-...oCIA	Mar 13, 2025	Apr 24, 2025	Daniele Ricciardelli	All

It is worth it to mention that these secret keys we might create are created and accessible only once, so it is important to save these keys and/or distribute them to your partners during the creation.

These keys can be quite distributed for different purposes, so if I decide to have another concurring key for another project, I can always set the limits myself from the dashboard. Highlight that depending on the project, we might have a cap on the usage of specific models, as shown below:

Organization budget

No budget set

Set budget

Rate limits

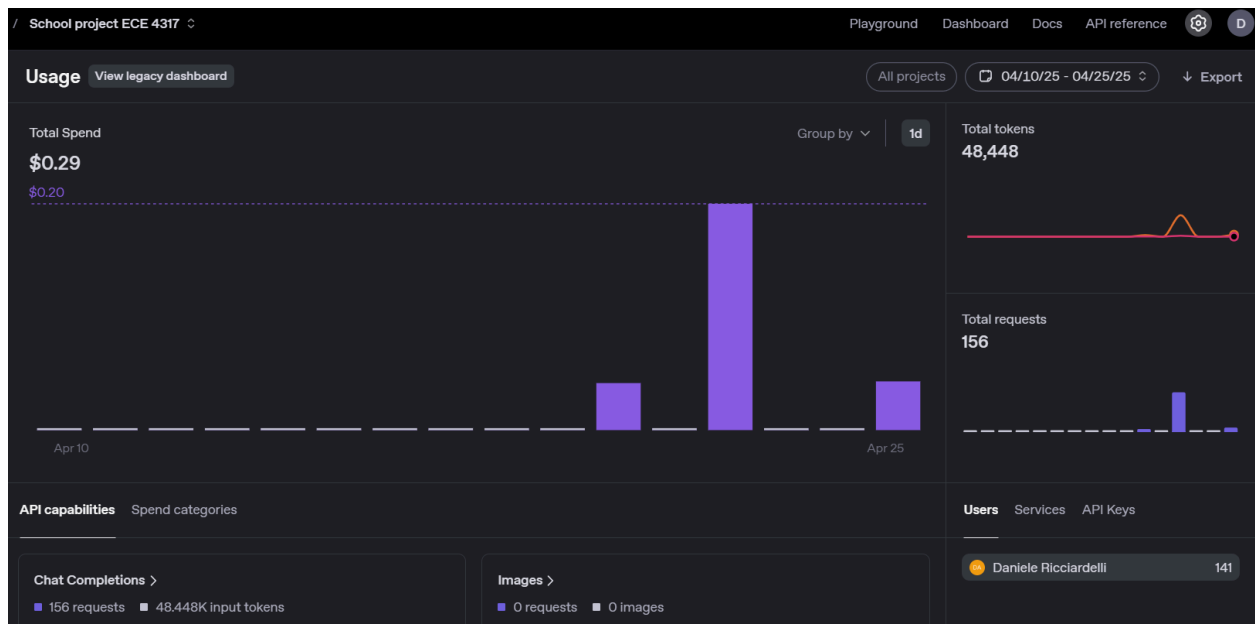
API usage is subject to rate limits applied on tokens per minute (TPM), requests per minute or day (RPM/RPD), and other model-specific limits. Your organization's rate limits are listed below.

Visit our [rate limits guide](#) to learn more about how rate limits work.

Note: Limits for specific model versions may vary, expand the table to see all models.

MODEL	TOKEN LIMITS	REQUEST AND OTHER LIMITS	BATCH QUEUE LIMITS
gpt-4.1	30,000 TPM	500 RPM	900,000 TPD
gpt-4.1-mini	200,000 TPM	500 RPM	2,000,000 TPD
gpt-4.1-nano	200,000 TPM	500 RPM	2,000,000 TPD
o4-mini	200,000 TPM	500 RPM	2,000,000 TPD
gpt-4o	30,000 TPM	500 RPM	90,000 TPD

And last but not least, the usage tab. In this tab we can see the amount of tokens used, requests, and budget spent. Quite interesting as it shows that after almost 50,000 tokens, it ran through barely to \$0.26. Obviously, this should be looked at differently from a student doing a project-research than a business model implementing OpenAI's API for their, for example, customer service. However, for the intended purposes, it turned out cheaper than we thought.



4. Python Scripting

To bring this project to life, we built a Python script that integrates several powerful libraries to create a seamless automation pipeline. Each library plays a key role in mimicking human

interaction, extracting information, and delivering intelligent responses. Many of these libraries are not installed in our computers, so we needed first to download them and install them, which can be easily done through the powershell (or CMD) command of:

```
pip install openai pytesseract pyautogui pillow flask opencv (6 different libraries)
```

The chain of thoughts we had to implement the idea were:

1. Access the API (openai)
 - a. flask will allow interaction with APIs
2. Access to images (pillow)
3. Decryption of images (pytesseract)
4. Access & Control for existing apps in Windows (pygetwindow)
5. Automation of screenshots (pyautogui)
6. Graphical User Interface (tkinter)
7. Determine if an image is a good match for onscreen elements (OpenCV)

Here's a breakdown of each component and how it functions in the overall system:

4.1 import openai

This is the keystone for this project, as this library will allow the communications with the OpenAI's model via API calls. For this project it allowed us to play with multiple chat GPT models, where we focused on 4o, 4o-turbo, o3, o3-mini, o3-mini-high, and recently 4.5.

Once the library is imported and installed, it is time to use it. However, in order to have access to it, we must have an API key, which we should've created in the step before.

```
client = openai.OpenAI(api_key="key_goes_here")
```

```
# Function to run GPT response
def get_chatgpt_response(prompt):
    response = client.chat.completions.create(
        model="gp                                # "gpt-4" to "gpt-4-turbo" , both work
        messages=[{"role": "user", "content": prompt}],# Sends input as the
prompt/message
```

```
        max_tokens=200 # Prevents utilization of too many tokens, we can adjust
this later at will
    )
    return response.choices[0].message.content
```

Here, we enable the **client** to read our API so it can process the needed queries.

The main heavy work happens in these lines of code that could also be found at their github [<https://github.com/openai/openai-python>], where we can choose the model. We heavily focused on gpt-4o for heavy reasoning and further use of tokens, and gpt-4-turbo for reasoning effectively (or so OpenAI claims), which we centered at the beginning. We also noticed that the responses were quite accurate when using 4o and 500-1000 tokens, but if less than 500, 4-turbo is quite better. At the end of the day, we ended up using only 500 or 1000 tokens as we had plenty of funds.

4.2 from PIL import Image

The PIL (pillow) library provides support for image processing tasks in Python, which will allow us in our script to read and save the image. This was used for testing purposes, but later on we found that there's no need to save the screenshot anymore, but just install the library itself.

4.3 import pytesseract

This library enables the Optical Character Recognition (OCR), which will read the image through Google's Tesseract engine. In other words, this will take the image and turn it into words so it can be processed as a prompt in our OpenAI implementation.

In order to configure this library, we first need to install it and set the system path by default like this:

```
pytesseract.pytesseract.tesseract_cmd = r"C:\Program Files\Tesseract-OCR\tesseract.exe"
```

4.4 import pygetwindow as gw

This enables the access and control of existing applications windows. The way that is used is so it can search the "Google Chrome" tab and select the first match. If there's multiple Chrome tabs, the first will always be the one that is at the top, in other words, at display.

```
windows = [win for win in gw.getAllWindows() if "Google Chrome" in win.title]
```

This can be changed if we wanted to work with other applications by simply changing “Google Chrome” by “Notepad” or if we wanted help with coding on Visual Studio, we could put “Visual”.

Once this is implemented, we can now use automation.

4.5 import pyautogui

This automates the movements, keyboard presses, and most importantly, the screen captures. Once the script detects the targeted tab with pygetwindow, we can implement what the script will detect as the screenshot

```
screenshot = pyautogui.screenshot(region=(left, top, width, height))
```

This will capture the targeted Chrome window ignoring the borders and taskbar.

With automation done, it will also be able to pass this to pytesseract now, with this line

```
text = pytesseract.image_to_string(screenshot, config='--psm 6 --oem 3')
```

Where `--psm6` will assume a single uniform block of text, and `--oem3` uses the OCR engine mode. Other modes like `oem1` and `oem2` work as intended, but are older versions.

4.6 import tkinter as tk

The purpose for this library is to implement simple GUI for the user to interact with where **Text()** will display chat history, **Entry()** accepts user input, and **Button()** will create a message to send specific text and re-run the script.

```
root = tk.Tk() # Sets Up the chatbox
root.title("GE stands for Generally Exhausting") # Tittle

chat_history = tk.Text(root, height=20, width=50) # Size of the chatbox
chat_history.pack() # History of the chat

entry = tk.Entry(root, width=50) # Chat entry
entry.pack() # Input field

send_button = tk.Button(root, text="Send", command=send_message) # Create send
button # Display button

rerun_button = tk.Button(root, text="Re-Run", command=rerun_script) # Re-run
button # Display button
```


4.6 OpenCV

PyAutoGUI will import OpenCV when running `pyautogui.locateOnScreen()`.

5. First results

The top screenshot shows a web browser window displaying a quiz from McGraw Hill. The quiz is titled "1 of 29 Concepts completed". The question is "What is the purpose of meditation?". The user's answer is "Gaining control over one's attention", which is marked as correct. The correct answer is also "Gaining control over one's attention". A ChatGPT window is open on the right, showing the user's input and the model's response: "ChatGPT: Koans [3rd]", "ChatGPT: breathing [4th]", and "ChatGPT: Answer: Gaining control over one's attention [1st]". The bottom screenshot shows the same web browser window, but the session has expired. A message box says "Session expired. Due to inactivity, you have been logged out." The user's answer is "ground", which is marked as correct. The correct answer is also "ground". The ChatGPT window on the right shows the user's input and the model's response: "ChatGPT: The correct answer is: 'It involves focusing on something that is repetitive or unchanging.' [3rd option]", "ChatGPT: Maharishi Mahesh Yogi [3rd]", and "ChatGPT: ground [4th]".

In here, we can see how in a test-like multiple choice online assignment this can be easily done as a side plugin, meaning the next step could be automation.

6. Automation

6.1 OpenCV

OpenCV is an open source computer vision library that can process images and is usually used for machine vision applications. It treats an image as a matrix of pixel values. With a pair of images as matrices it can compare them and decide how similar they are. This functionality is useful for finding specific elements on screen.

6.2 Implementation

The automation script is being handled by PyAutoGUI and OpenCV to compare an image with onscreen elements. PyAutoGUI loads an image of the answer field (“answer.png”), takes a screenshot of the entire screen, and uses OpenCV’s template matching to find the answer box in the screenshot. If an onscreen element appears to be an answer box with a reasonable degree of certainty, then the answer number given by ChatGPT is checked. The correct answer is then selected by clicking within the answer boxes using a predefined pixel offset. The code is designed to solve quiz-maker.com quizzes. By replacing “answer.png” with a sample from another site and adjusting the pixel offsets, it will be able to solve quizzes from other sites as well.

The code in its current state can only answer single answer multiple choice questions and must be told to rerun for each new question. When the code is finished it will be able to detect if a question is multiple choice, many multiple choice, true or false, or fill-in the blank. If it is fill-in the blank it will use machine vision to find and select the text box, then type the answer. True/false will use the same method as multiple choice but with only two answer options and many multiple choice will loop the multiple choice process.

In the code below “location” stores the position of the answer space on the screen. The confidence level has been set to 20% because OpenCV is uncertain of a match when the text of the answers is different from the sample image. “answer” is the string that ChatGPT returned as the answer; the answer number is located in the fourth position from the end of the response.

6.3 Code

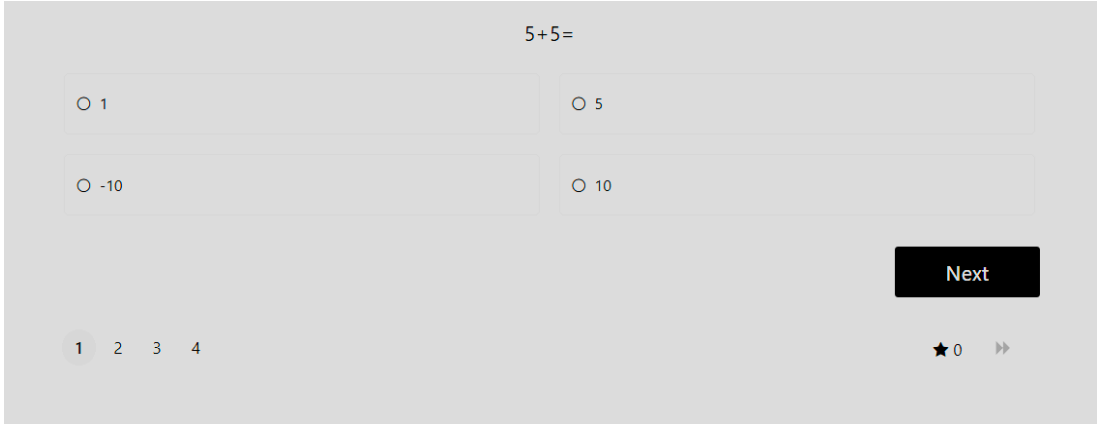
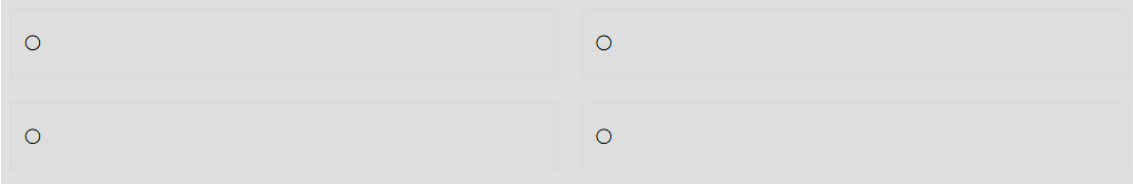
```
click_answer(chat_response) # Call automation
function

def click_answer(answer):
    # Find the checkbox, if the image is zoomed it might not work
    location = pyautogui.locateOnScreen('answer.png', confidence=0.2)
```

```
x = location.left
y = location.top
w = location.width
h = location.height

number match answer[-4]:                                # Checks the answer

    case "1":
        pyautogui.click(x + 28, y + 39)                # Click checkbox
    case "2":
        pyautogui.click(x + 540, y + 39)
    case "3":
        pyautogui.click(x + 28, y + 122)
    case "4":
        pyautogui.click(x + 540, y + 122)
    case _:
        print("Error finding answer!")
        exit()
```



On top is the “answer.png” being used to find the answer boxes. on bottom is an example question page.

7. Plugin thoughts

As an idea, this project turned out to be a possible future project to pick up again. Now that we have some extra budget to use with OpenAI's, our next milestone could be to make a plugin that could automate the same thing as we have but in chrome as an assistant, or as a type of Co-Pilot as Microsoft Edge has.

8. Conclusion

Using OpenAI's ChatGPT-4o model's API is an effective way to get help on tests. Using PyTesseract, PyAutoGUI, Pillow, and OpenCV allows for automation of the help requests and can even allow automation of the entire test.

This project was a great experiment to learn what can be done with AI to help students with their school tasks.