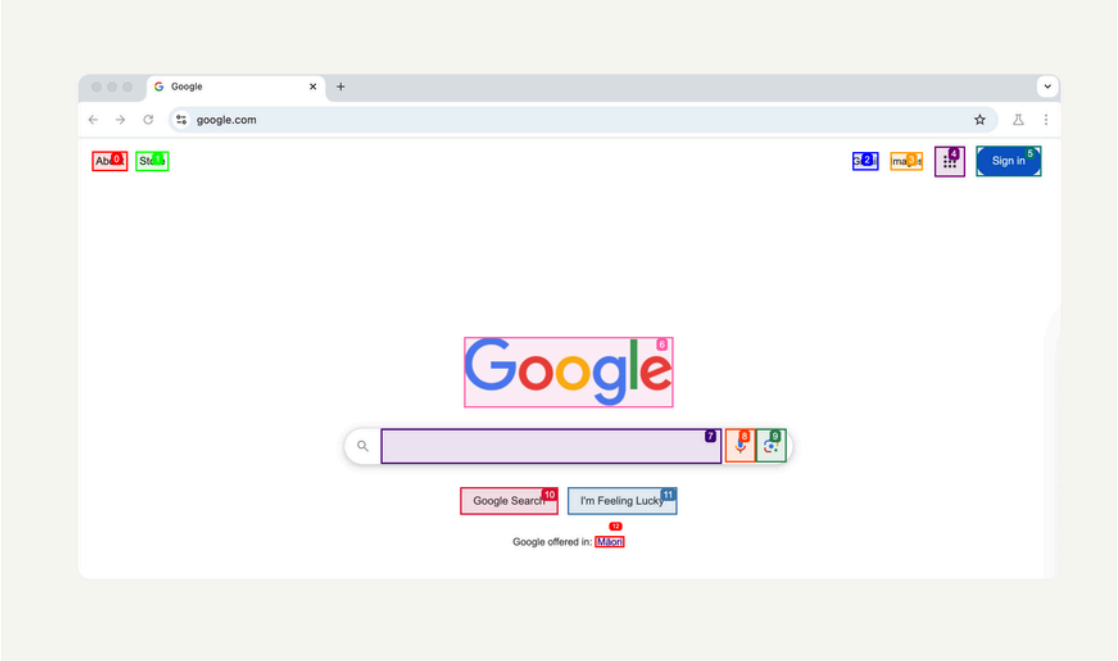
Kareshma
Rajaananthapadmanaban

Automate Browser with AI Agents

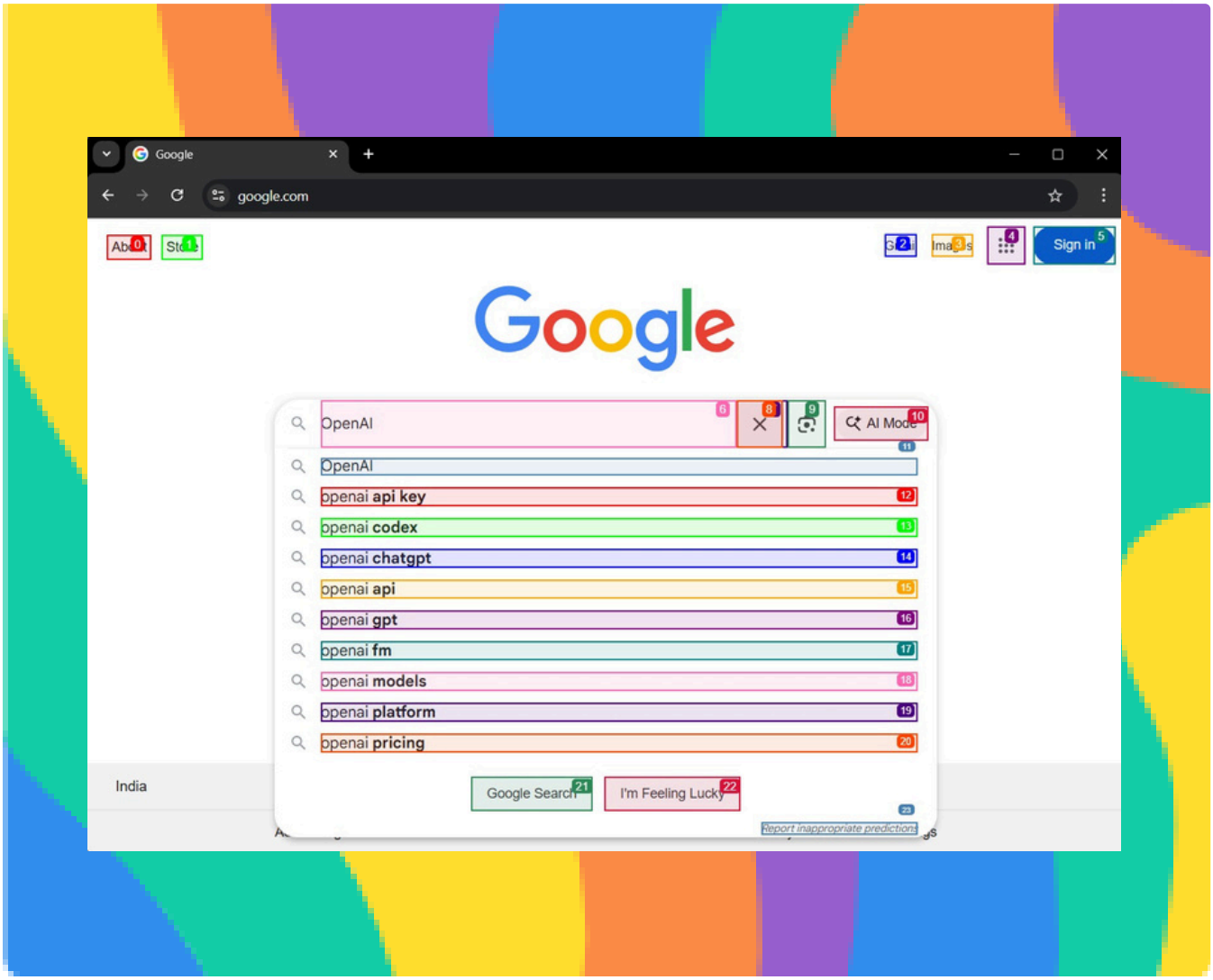
Create AI agents to search for information, fill out forms, explore websites, and more!



nextwork.org

Automate Your Browser with AI Agents

Kareshma Rajaananthapadmanaban



Introducing Today's Project!

In this project, I will demonstrate how to create **AI-powered agents** that automate browser tasks like searching, filling out forms, and navigating websites. I'm doing this project to learn how to reduce repetitive manual work using automation tools. Itś' a hands-on way to explore browser control with AI and improve productivity through smart agents.

Tools and concepts

Services I used were Amazon Q **WebUI,** local **Chrome instance**, and `**.env**` configuration. Key concepts I learnt include using remote debugging with Chrome, setting environment variables for browser automation, managing authenticated sessions securely, and resolving agent-browser conflicts during automation tasks.

Project reflection

This project took me approximately 2.5 hours. The most challenging part was fixing the "**No module" error**, which required editing parts of `**webui.py**` to get it running. I also hit a browser control error turns out, Chrome keeps background processes that block agent access. The root cause was Chrome's default behavior, which I solved by launching a **debug-enabled instance.** It was most rewarding to see the agent finally execute real browser actions.

I did this project to learn how to run **real browser automation** using AI agents and test local workflows with authenticated sites. It aligned perfectly with my goal of exploring practical agent use cases. I now understand how to control browser sessions securely and troubleshoot common setup issues like Chrome blocking agent access.

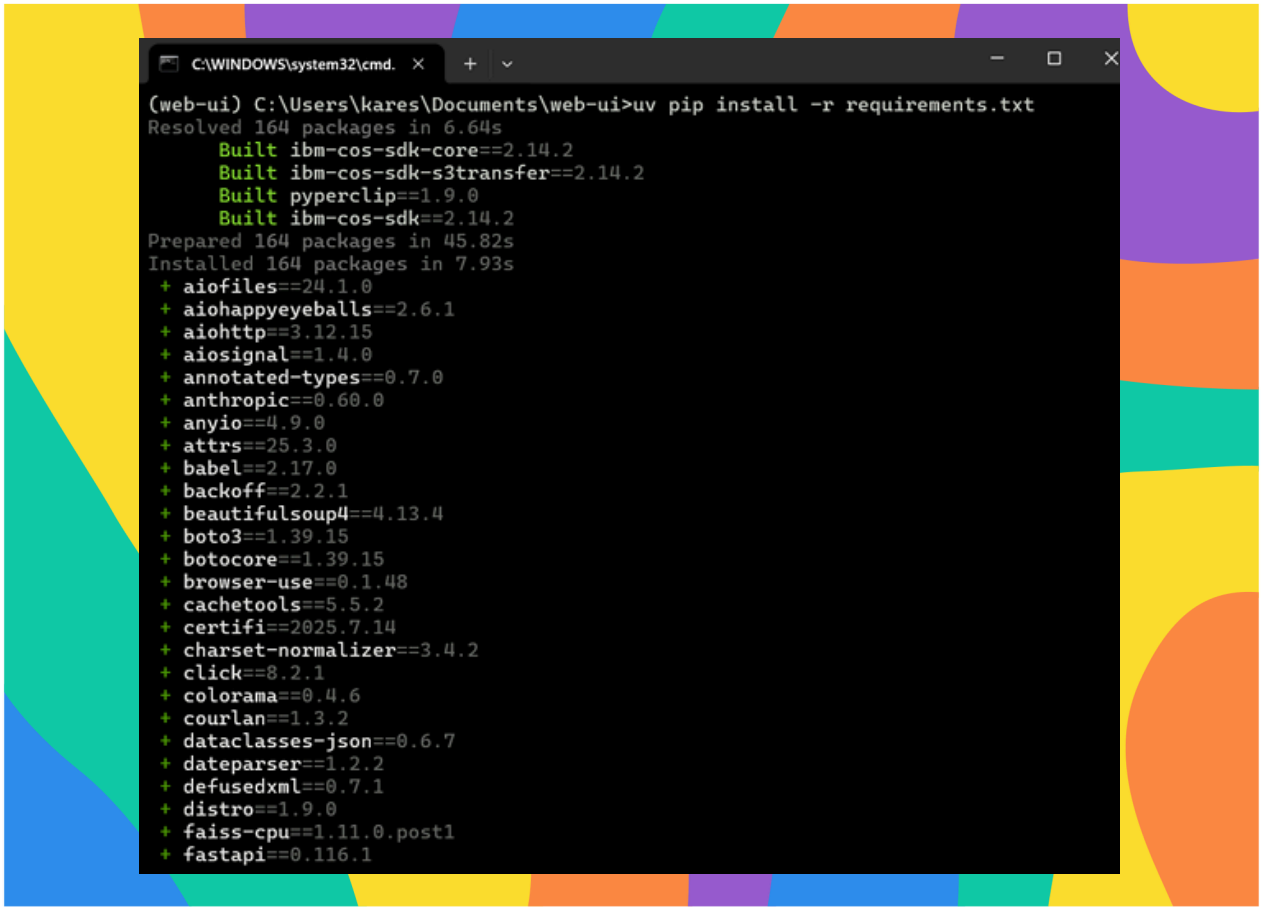
Development Environment

In this step, I will install **Python, Pip, UV,** and **Git** because these tools are required to run and manage WebUI. Python runs the core of the app, Pip and UV handle package installations, and Git helps me fetch the latest WebUI code. Setting this up ensures my environment is ready for browser automation.

I installed Python, Pip, UV, and Git because they are essential for running and managing WebUI. Python powers the backend, Pip and UV handle package installations, and Git lets me **clone** the latest **WebUI source code**. I also checked for PATH issues and upgraded Pip and UV to avoid compatibility problems.

In this step, I will download WebUI's source code from GitHub, set up a **Python virtual environment,** and run the app because WebUI isn't a pre-built application. Running it locally gives me full control, and the virtual environment keeps dependencies isolated, ensuring WebUI runs without interfering with other Python projects.

WebUI is a visual interface that lets me interact with **Browser Use** without writing code. To retrieve WebUI's code, I need to clone its GitHub repository using Git, navigate into the project folder, and set up a virtual environment. Once the dependencies are installed and the server is running, I can **access WebUI** in my **browser** to start automating tasks.



Terminal showing the packages

I set up a virtual environment by running `uv venv --python 3.12.3` and activating it with `.venv\Scripts\activate` (or the Windows equivalent). A virtual environment is helpful for isolating dependencies, avoiding version conflicts, and keeping the project clean and reproducible across different setups.

While running python webui.py, I encountered a ModuleNotFoundError for distutils, which is deprecated and not included by default in some Python installations. Initially, I tried resolving it by importing strtobool directly from distutils.util, but the error persisted, even after ensuring setuptools was installed.

```
 + zstandard==0.23.0

(web-ui) C:\Users\kares\Documents\web-ui>playwright install

(web-ui) C:\Users\kares\Documents\web-ui>playwright --version
Version 1.54.0

(web-ui) C:\Users\kares\Documents\web-ui>copy .env.example .env
Overwrite .env? (Yes/No/All): yes
        1 file(s) copied.

(web-ui) C:\Users\kares\Documents\web-ui>python webui.py --ip 127.0.0.1 --port 7788
Traceback (most recent call last):
  File "C:\Users\kares\Documents\web-ui\webui.py", line 4, in <module>
    from src.webui.interface import theme_map, create_ui
  File "C:\Users\kares\Documents\web-ui\src\webui\interface.py", line 5, in <module>
    from src.webui.components.browser_settings_tab import create_browser_settings_tab
  File "C:\Users\kares\Documents\web-ui\src\webui\components\browser_settings_tab.py",
 line 2, in <module>
    from distutils.util import strtobool
ModuleNotFoundError: No module named 'distutils'

(web-ui) C:\Users\kares\Documents\web-ui>
```

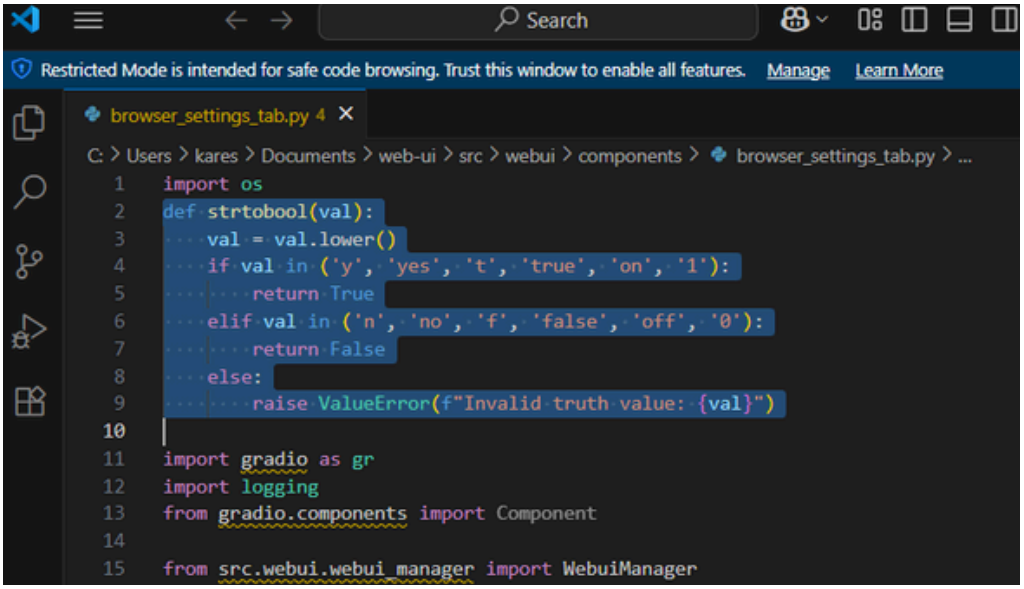
```
(web-ui) C:\Users\kares\Documents\web-ui>pip install setuptools
Requirement already satisfied: setuptools in c:\users\kares\appdata\local\programs\py
thon\python312\lib\site-packages (80.9.0)

(web-ui) C:\Users\kares\Documents\web-ui>python webui.py --ip 127.0.0.1 --port 7788
Traceback (most recent call last):
  File "C:\Users\kares\Documents\web-ui\webui.py", line 4, in <module>
    from src.webui.interface import theme_map, create_ui
  File "C:\Users\kares\Documents\web-ui\src\webui\interface.py", line 5, in <module>
    from src.webui.components.browser_settings_tab import create_browser_settings_tab
  File "C:\Users\kares\Documents\web-ui\src\webui\components\browser_settings_tab.py"
, line 2, in <module>
    from setuptools._distutils.util import strtobool
ModuleNotFoundError: No module named 'setuptools'

(web-ui) C:\Users\kares\Documents\web-ui>
```

To fix it, I inspected the error trace and **edited** the problematic file browser_settings_tab.py. Instead of relying on distutils, I replaced the import line with a custom strtobool function defined locally. This manual patch bypassed the deprecated module and resolved the import error.

To fix it, I inspected the error trace and edited the problematic file browser_settings_tab.py. Instead of relying on distutils, I replaced the import line with a **custom strtobool function** defined locally. This manual patch bypassed the deprecated module and resolved the import error.



Editing Browser_settings_tab.py

After saving the file, I re-ran the command and successfully launched WebUI at http://127.0.0.1:7788. The app booted without issues, confirming that the custom fix worked and the server was running locally.

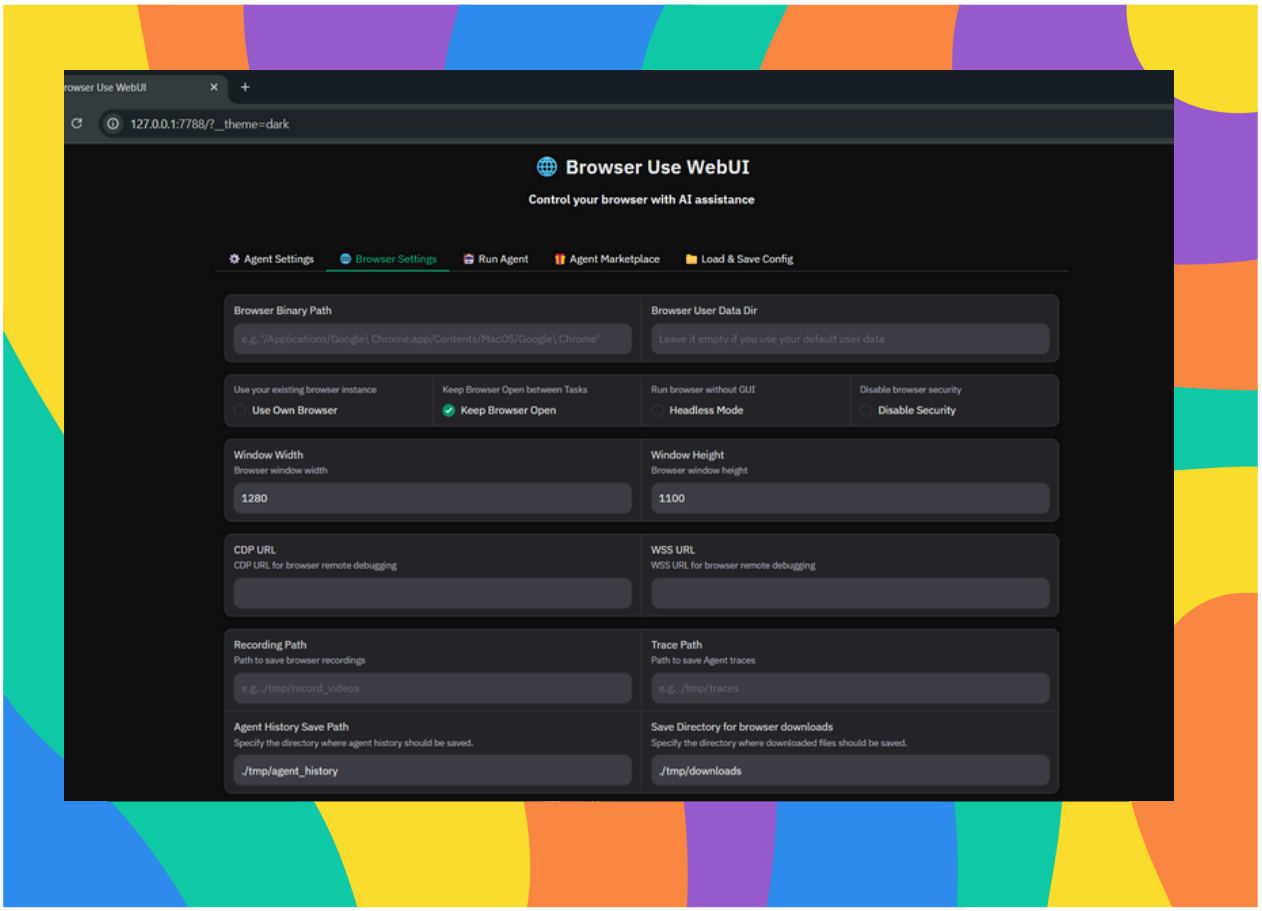
```
(web-ui) C:\Users\kares\Documents\web-ui>python webui.py --ip 127.0.0.1
--port 7788
* Running on local URL:  http://127.0.0.1:7788

To create a public link, set `share=True` in `launch()`.
```

Configuring My API

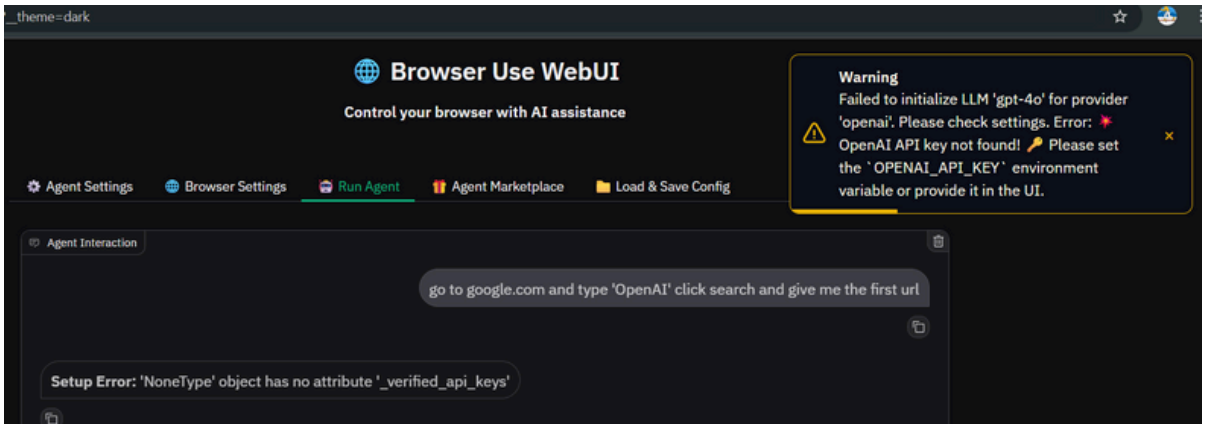
In this step, I will run WebUI, handle any errors that come up, and configure it with an **API key** from **Google AI Studio** because WebUI needs access to a language model to power the browser agent. Storing the API key securely ensures the application runs safely without exposing sensitive credentials.

An API key is needed because it **authorizes WebUI** to communicate with **external AI services** like Gemini via Google AI Studio. It helps WebUI by allowing the agent to send prompts and receive intelligent responses, powering the decision-making behind each browser automation task. Without it, WebUI can't access any AI model to function.

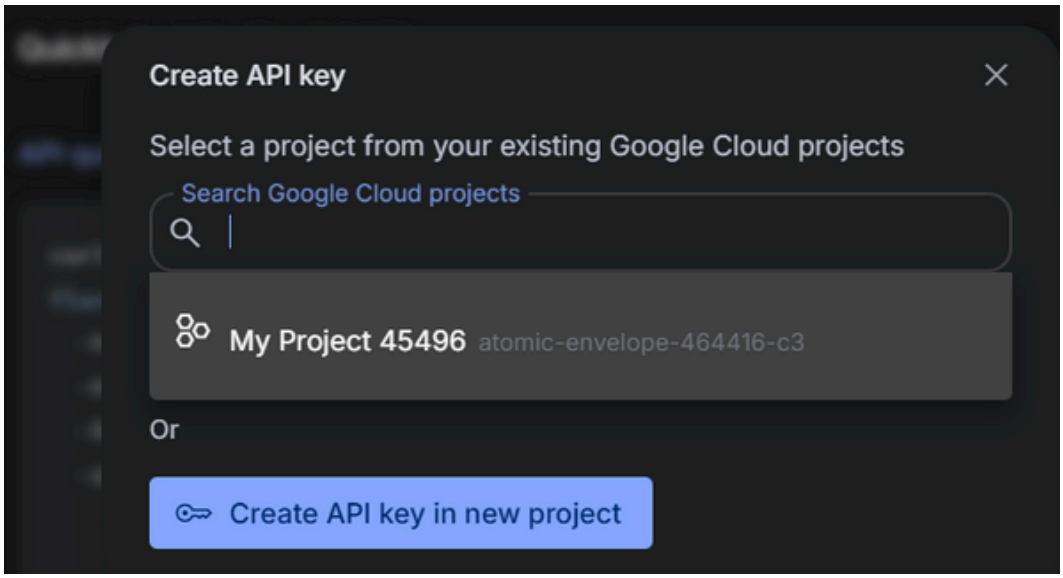


Browser WebUI running at Local host

When I clicked Run Agent with the default prompt, I hit an error saying the OpenAI API key was missing. Since I planned to use Google's free-tier model instead, I went to Google AI Studio, generated a Gemini API key, and copied it. WebUI supports Gemini, so I pasted this key into the **LLM Configuration** tab. This let me securely connect WebUI to Google's AI model without exposing credentials or depending on OpenAI.

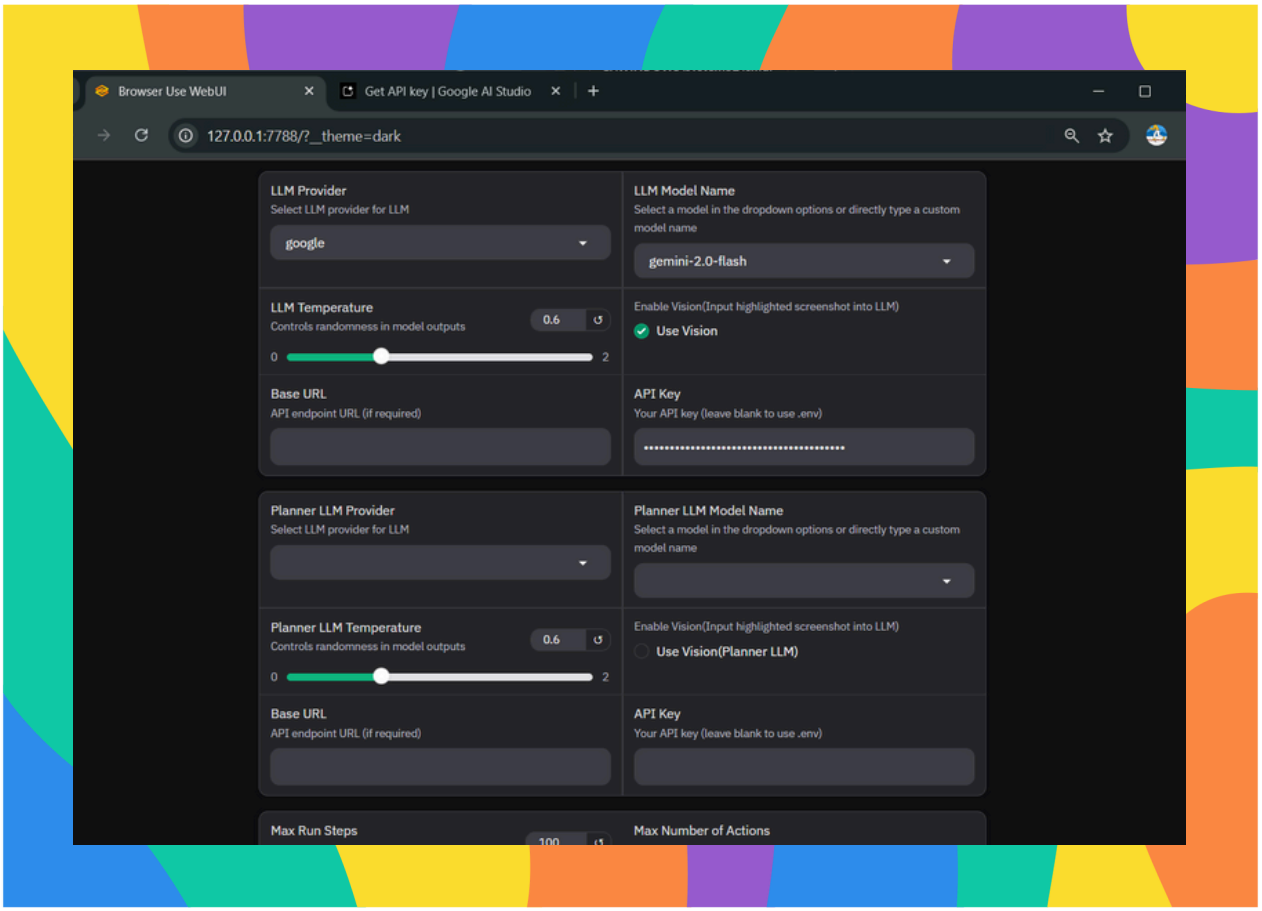


Error API Key NOT FOUND



I set up my API key by generating a new one in Google AI Studio. Then I **configured WebUI** to use it by entering the API Key in the LLM Configuration tab. I also switched the AI Model from OpenAI (default) to Gemini (Google AI Studio's model).

After launching WebUI, I explored its tabs to understand the full setup. Each tab has a specific role: Agent Settings controls the agent's browsing limits, LLM Configuration connects the agent to an AI model, **Browser Settings** lets me choose which browser to automate, and Run Agent is where I define what task the agent should perform. Other tabs like **Results** and **Recordings** are useful for reviewing the **agent's activity.**



LLM Configuration settings

Running The Agent

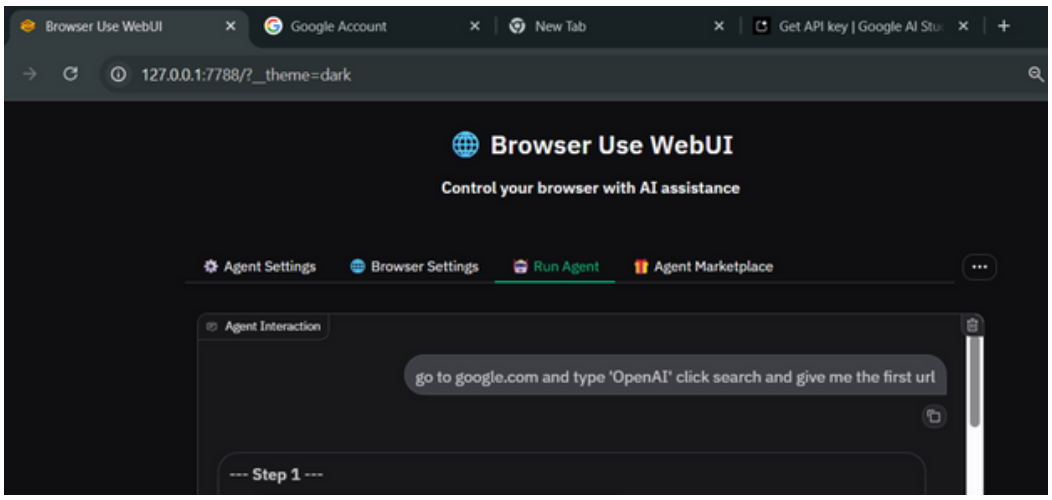
In this step, I will run my AI agent using WebUI and observe how it completes a **browser task** because this is the core of browser automation. Watching the agent navigate, search, and interact with the page helps me validate that everything is configured correctly and confirms that the AI model is successfully driving the agent's actions.

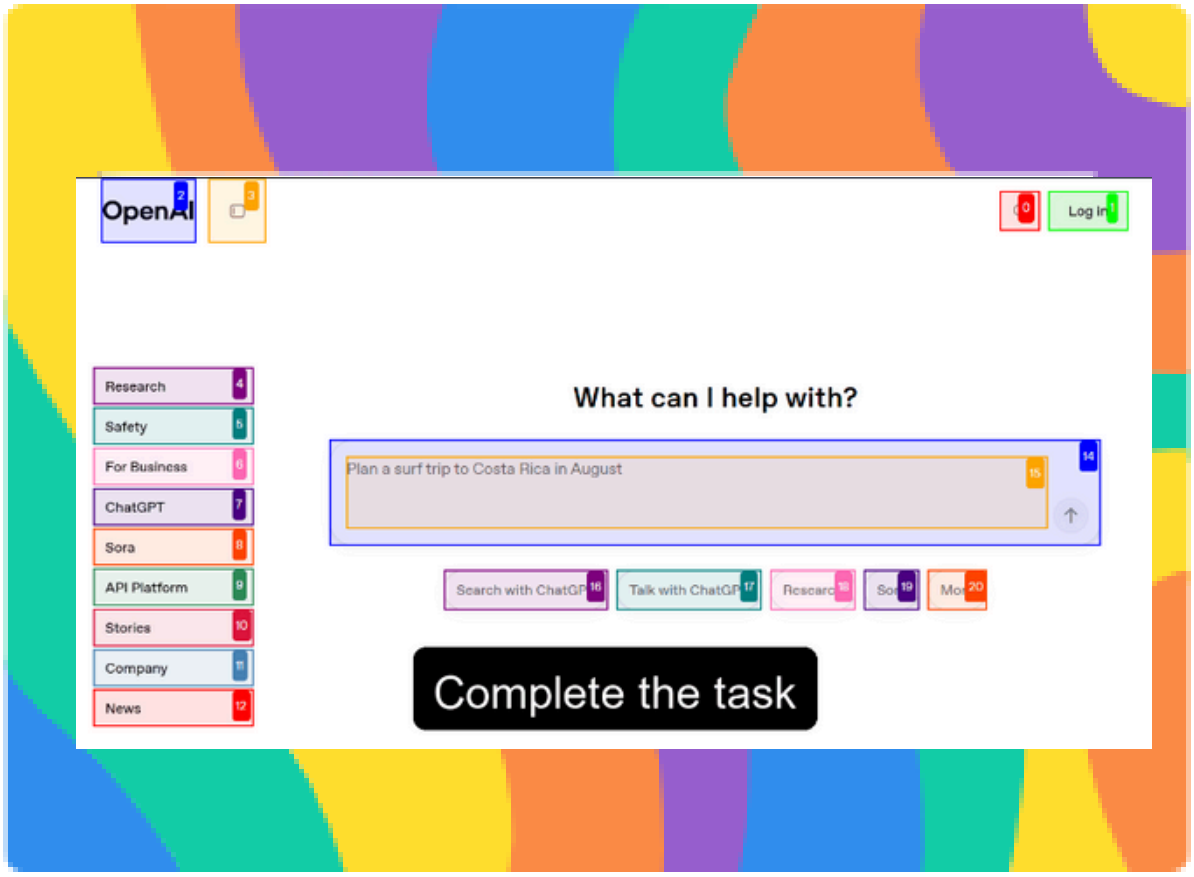
My first agent's task was to go to google.com, search for "OpenAI", and return the first URL. The agent by default launches **Chromium**, which is a fast, open-source browser engine used by Chrome and others. It handled the task smoothly, loaded the page, highlighted elements using Playwright, and clicked through automatically completing the task without manual input through AI-powered decision-making.

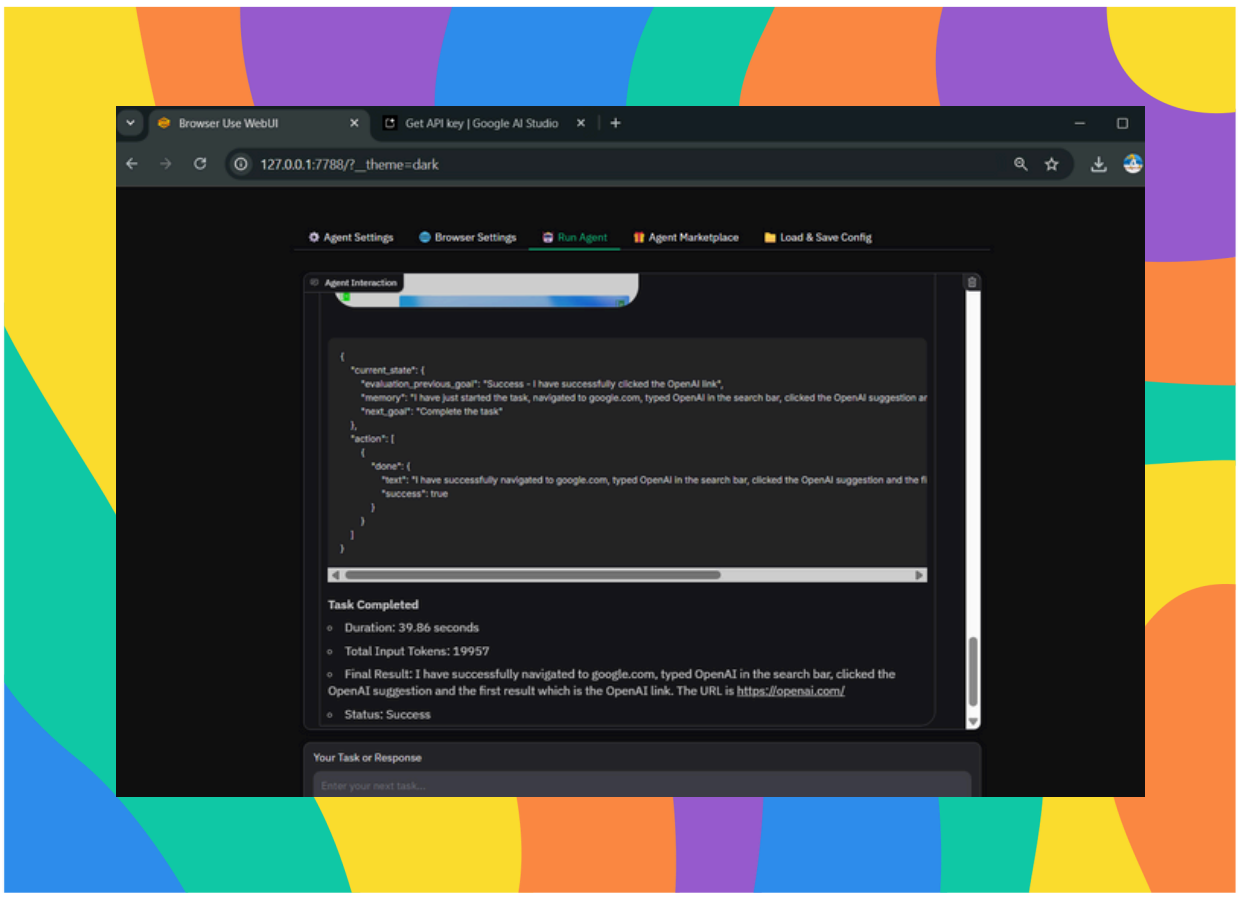
The **agent interacts** with the browser by using **Playwright** to scan the page, highlight clickable elements, and send them to the Gemini AI model for analysis. I can see this when **colored boxes** appear around **links** and **buttons** in real time, showing what the agent is evaluating and clicking as it follows the task prompt step by step.

The **Results tab** shows a **log of the agents** actions, including the steps it took, decisions made, and the final output in this case, the first URL found for "OpenAI". This is helpful because it lets me verify if the agent completed the task correctly and provides transparency into how the AI interpreted and interacted with the page.

Running The Agent







WebUI agent interaction

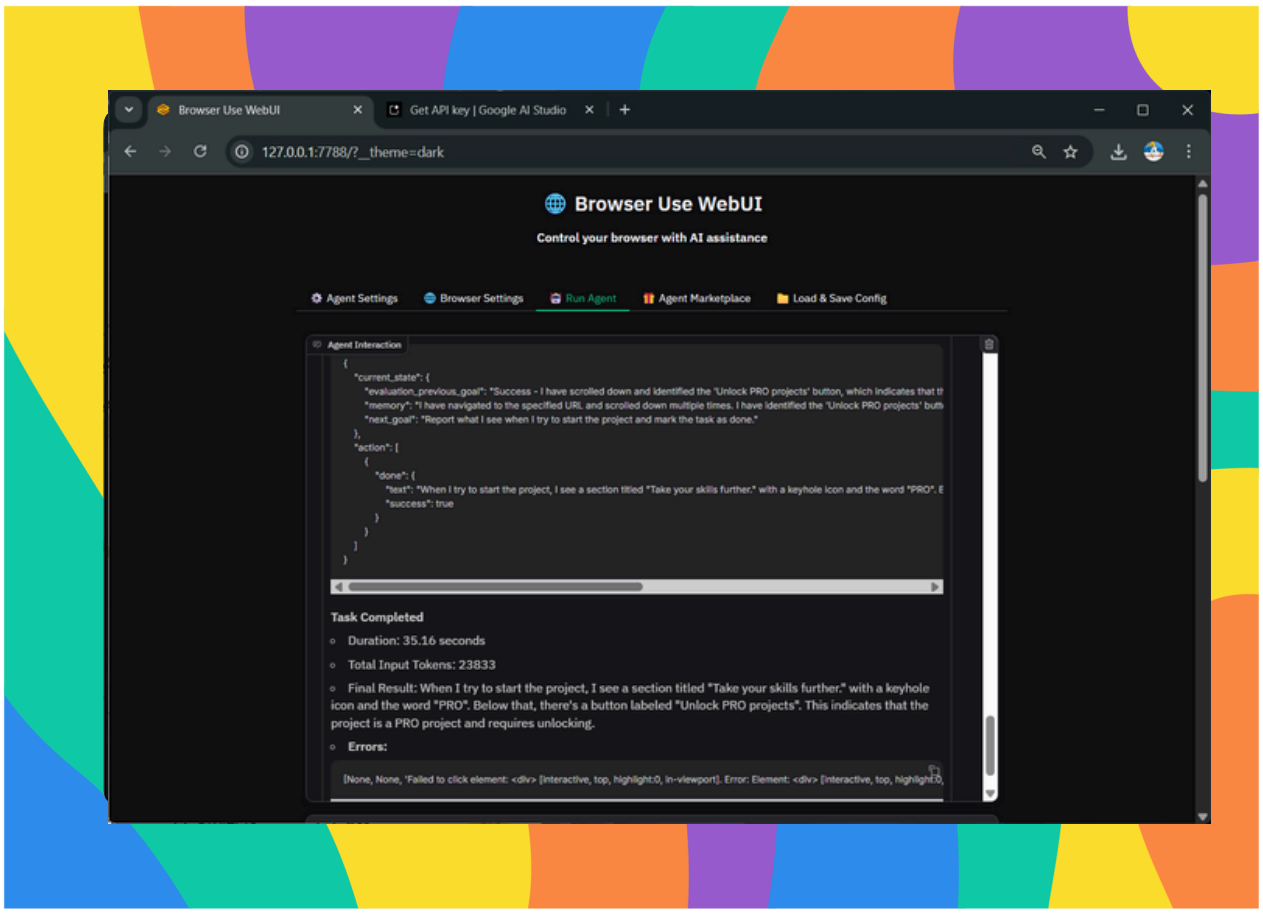
Advanced Navigation

After confirming the agent's basic task worked, I moved on to a more advanced prompt this time testing how the agent responds to **interaction-based scenarios**. In the Run Agent tab, I asked it to visit a specific page on **NextWork** and **report** what it sees when trying to start a project.

This unlocked a kind of "secret mission," where the agent needed to go beyond just clicking links it had to understand the page's layout, **detect access restrictions,** and report back the outcome. The agent navigated successfully, scrolled through the content, and accurately responded that PRO access is required to start the project. It showed how AI-driven browser automation can help test user flows or **uncover UI blockers** in real-world web apps.

Advanced Navigation

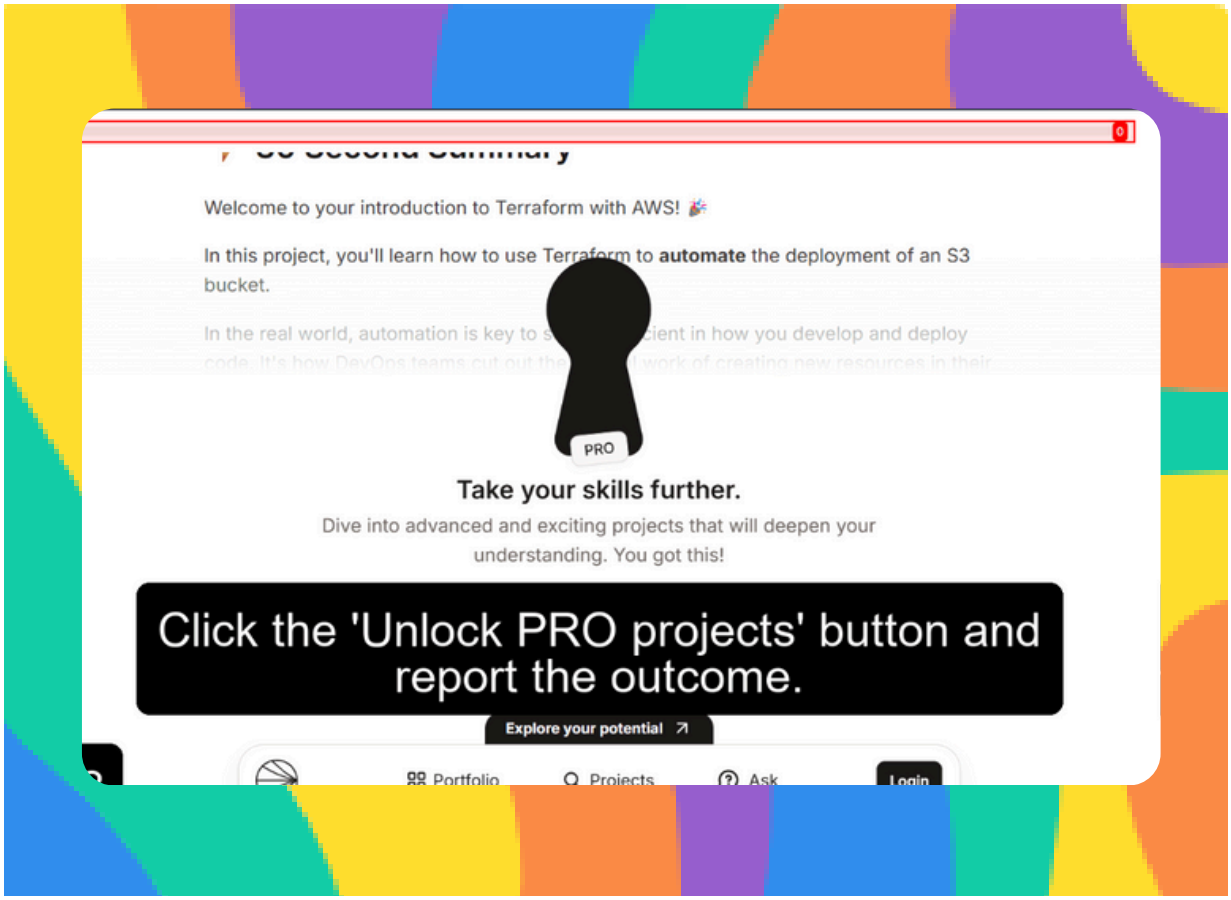
I also tried telling the agent to visit a specific project page on NextWork and see what happens when starting the project. It reported that access was restricted and starting the project required a PRO membership. This confirmed that the agent could handle **multi-step tasks** and **recognize access limitations** in real-world websites.



Agent's response about PRO access.

Advance Navigation

Visit https://learn.nextwork.org/projects/aws-devops-terraform1. Scroll down and tell me what you see when you try to start the project.



Debugging and Logs

In this step, I will experiment with more **complex prompts** and learn how to troubleshoot common agent issues because **prompt clarity directly affects** how well the **agent performs tasks**. Understanding how to guide the agent through multiple steps and fix errors ensures smoother, more reliable browser automation.

When my **agent** encounters an error, **three common reasons** are:

- The task description is vague or lacks a clear expected output,
- The website layout is too complex for the agent to navigate or understand properly, or
- The site has bot protection that blocks automated browsing.

These issues prevent the agent from completing the task as intended.

The **terminal logs** show a detailed, step-by-step trace of the agentś' actions like navigating to pages, locating elements, clicking, or extracting data. This helps me understand exactly what the **agent** is doing **behind the scenes**, how it interprets the task, and where it might have run into issues or succeeded during execution.

When I asked the agent to extract Instagram and Substack links, it failed its first attempt because the **prompt didn't clearly state** what output was expected or how to find the links. I improved it by specifying the exact elements to extract, the format to return them in, and allowing tab navigation. This made the agent's job clearer and it succeeded.

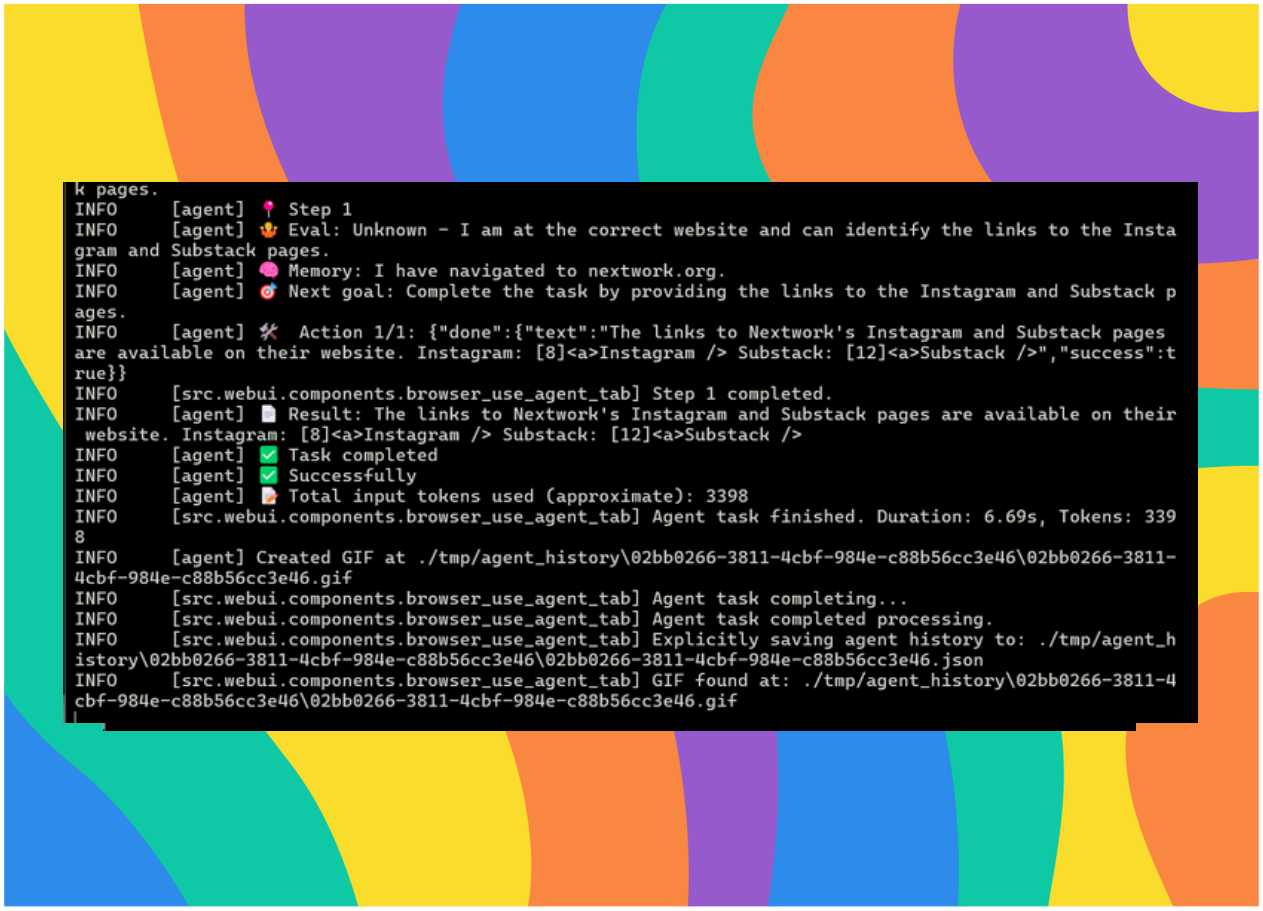
Debugging and Logs

Before the improved prompt, the agent attempted to access the project page but ran into a paywall requiring PRO access. Although it successfully navigated to the correct URL and scrolled the page, it didn't perform the expected action of starting the project. This revealed a key limitation **not all** websites will **allow automation-based interaction** without authentication or subscription.

Once the secret mission was unlocked, it became clear how browser agents interpret task descriptions. When no explicit instruction or output format is given, the agent might return vague or incomplete results, or fail entirely.

By refining the prompt to **specify** the **expected output** (e.g., full URLs separated by newline), the task became easier for the agent to understand and execute. This step emphasized the importance of being **precise** and **directive** in prompt design to avoid miscommunication between you and the AI.

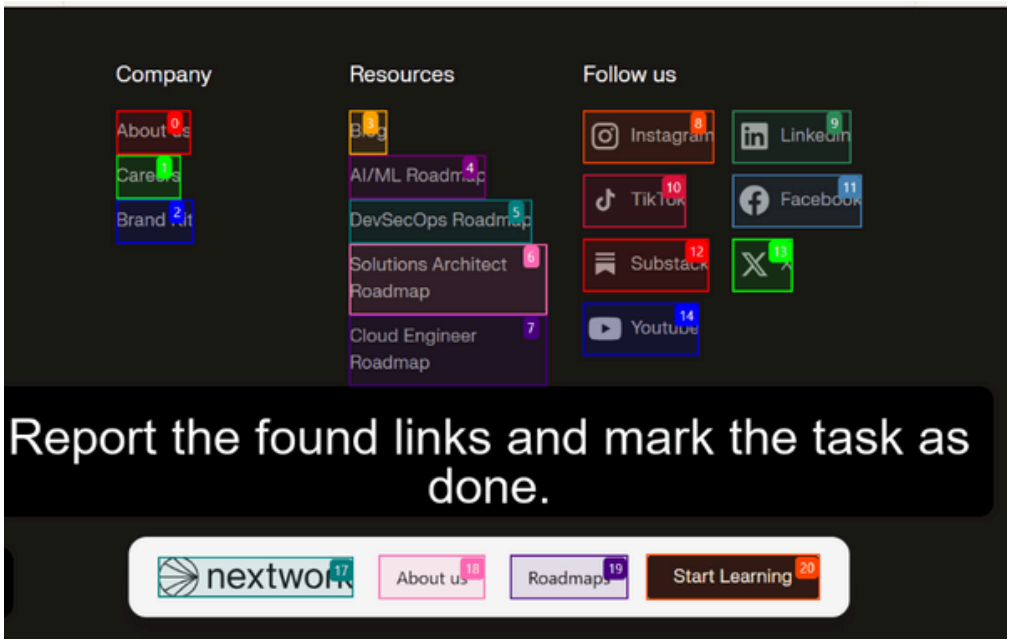
> The exercise also showed how errors **aren't** always due to **technical faults** sometimes it's just **unclear instructions.** You don't need to be overly verbose, but structure matters: include what, where, and how. That small shift in wording was the difference between failure and success.



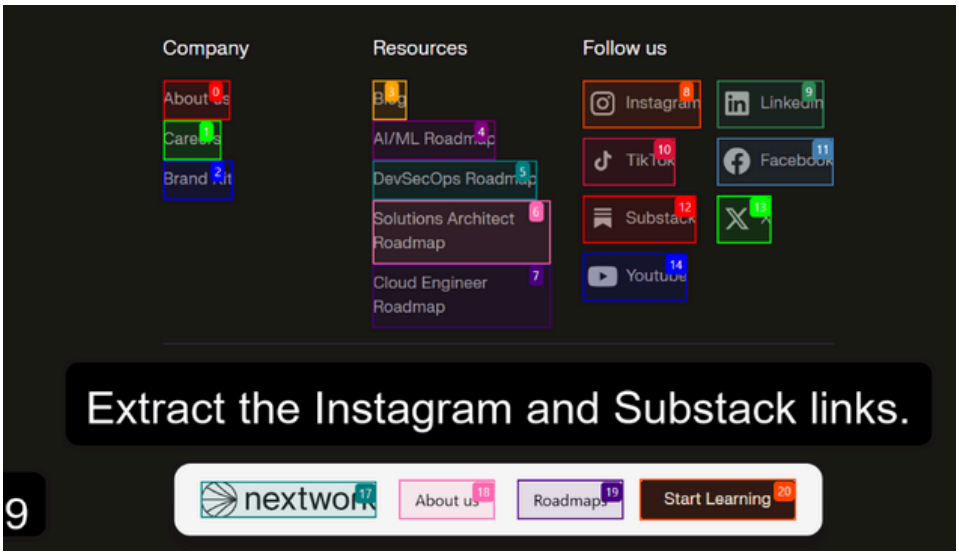
Terminal showing the WebUI agent's logs.

Complex Prompts

Go to nextwork.org and find the links to their Instagram and Substack pages.

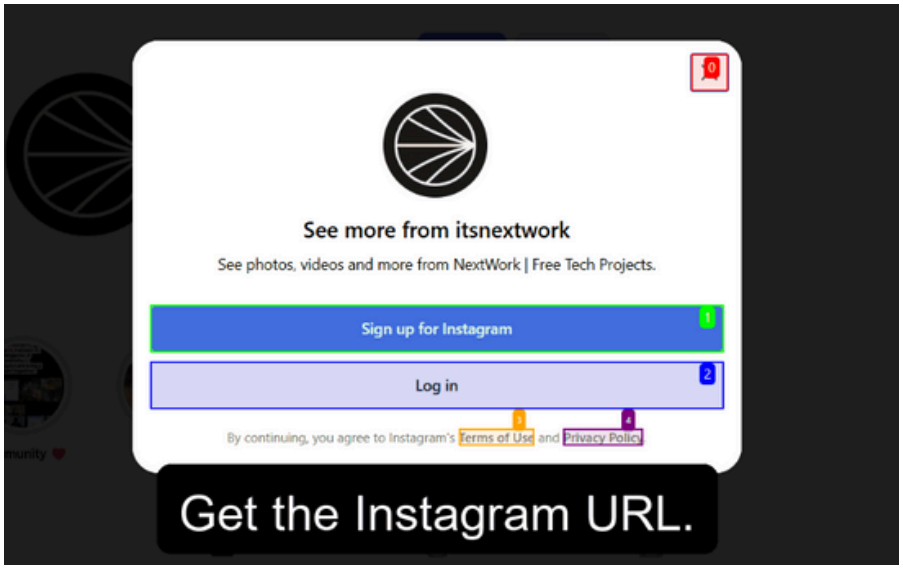


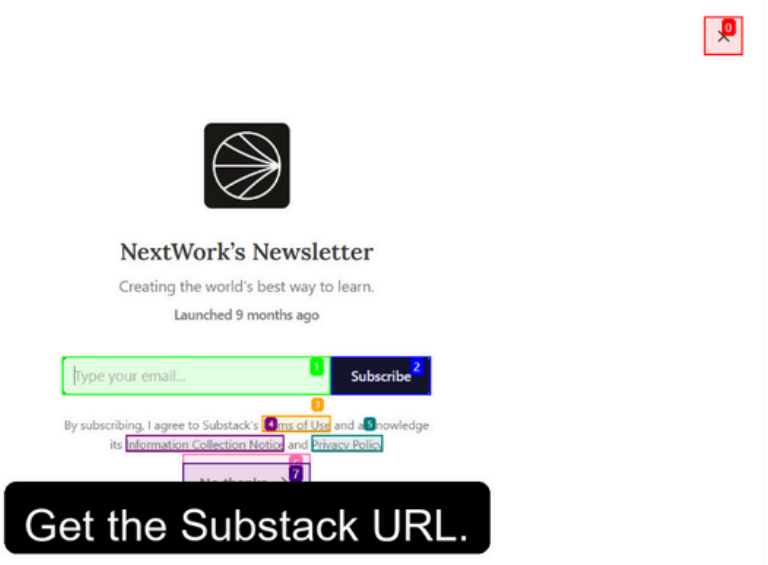
Go to nextwork.org and extract the full URLs for their Instagram and Substack pages. Both URLs are at the bottom of the page. Return only the URLs, separated by a newline. You are allowed to open new tabs to navigate to the Instagram and Substack pages.



Complex Prompts

Instagram and Substack pages.





Advanced Browser Configuration

In this project extension, I will configure the agent to **use my own browser** instead of the default chromium one. Using my own browser is useful because it gives the agent access to existing logins, sessions, and cookies, letting it interact with websites as if it were me, meaning it can **access content behind authentication** walls like LinkedIn, GitHub, or Substack without having to log in from scratch. This allows the agent to perform tasks that require authentication, like accessing LinkedIn or personalized dashboards.

To use your own browser in WebUI, you have to update the `.env` file with your local Chrome setup. The **`BROWSER_PATH`** setting tells WebUI where to find the Chrome executable on your machine, while the **`BROWSER_USER_DATA`** setting tells it which user profile to load this lets the agent access saved sessions and logins. I also changed **`KEEP_BROWSER_OPEN`** to `false` and set **`USE_OWN_BROWSER`** to `true` so WebUI knows to launch and control my personal browser instead of a temporary one.

I chose to test my agent with the task: `Go to LinkedIn and search "nextwork" "ai" "project". Ensure that all search terms are enclosed in double quotes. Your task is to like the first post.` The agent was able to perform this action directly in my **personal Chrome browser** because I had already logged into LinkedIn. This works only if all Chrome windows are fully closed before launching WebUI, ensuring a clean session. I restarted WebUI by running **`python webui.py --ip 127.0.0.1 --port 7788`** in the terminal, then reloaded `http://127.0.0.1:7788` in my browser.

Advanced Browser Configuration

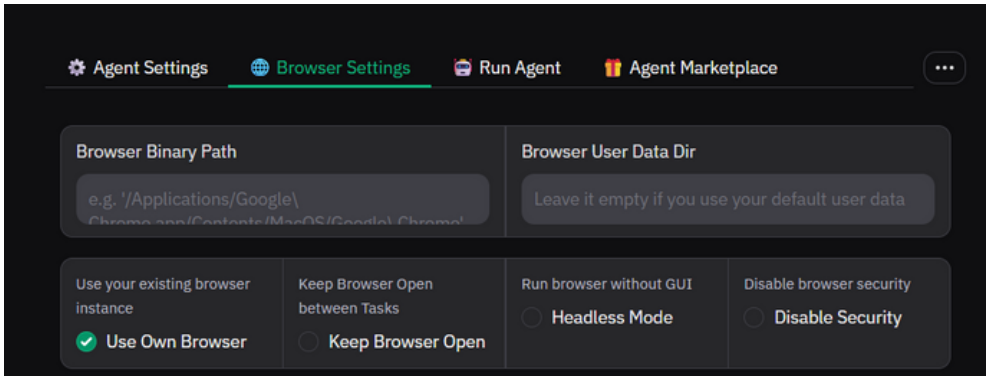
Remember to replace YOUR_USERNAME in the path (C:\Users\YOUR_USERNAME\AppData\Local\Google\Chrome\User Data) with your actual Windows username. For example, I replaced it with **kare**

This setup lets the **agent access authenticated** websites and perform real actions, like liking a LinkedIn post or sending an email, without getting blocked by login walls.

```
# Browser settings
BROWSER_PATH="C:\Program Files\Google\Chrome\Application\chrome.exe"
BROWSER_USER_DATA="C:\Users\kares\AppData\Local\Google\Chrome\User Data"
BROWSER_DEBUGGING_PORT=9222
BROWSER_DEBUGGING_HOST=localhost
# Set to true to keep browser open between AI tasks
KEEP_BROWSER_OPEN=false
USE_OWN_BROWSER=true
BROWSER_CDP=
# Display settings
# Format: WIDTHxHEIGHTxDEPTH
RESOLUTION=1920x1080x24
```

Update the .env file

In Browser Settings, I enabled **Use Own Browser** to make the agent use my existing Chrome profile with cookies and login sessions.



Advanced Browser Configuration

I test my agent with the LinkedIn prompt: Go to LinkedIn and search "nextwork" "ai" "project". Your task is to like the first post. This required the agent to interact with a logged-in session, leveraging my existing browser profile. However, I initially ran into an error **WebUI couldn't control** the browser properly.

This happened because **Chrome** was **already running** in the background, and the agent was attempting to launch a new instance over it. Essentially, two processes were trying to steer the same browser, causing a conflict.

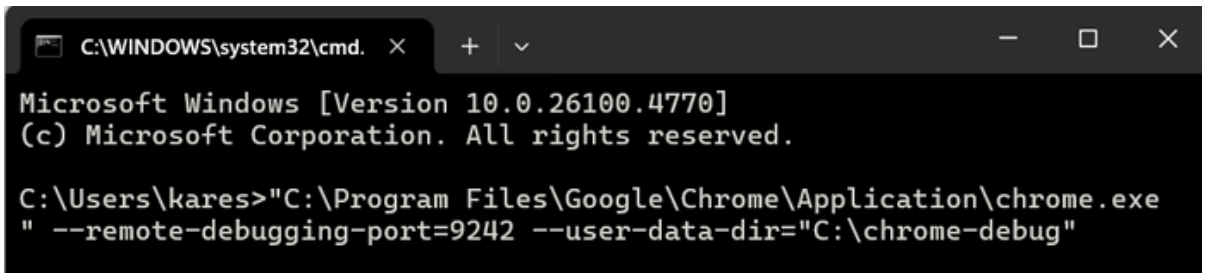
```
Create a new document.
Insert three motivational quotes.
Save the document.
Print it as Pdf.
INFO     [agent] 📍 Step 1
INFO     [browser] 🌐  Storing Browser Profile user data dir in: C:\Users\kares\AppData\Local\Google\Ch
rome\User Data
ERROR    [browser] ❌  Failed to start a new Chrome instance: BrowserType.connect_over_cdp: connect ECO
NNREFUSED ::1:9242
Call log:
  - <ws preparing> retrieving websocket url from http://localhost:9242

ERROR    [browser] Failed to initialize Playwright browser: To start chrome in Debug mode, you need to
close all existing Chrome instances and try again otherwise we can not connect to the instance.
ERROR    [browser] ❌  Failed to create new browser session: To start chrome in Debug mode, you need to
 close all existing Chrome instances and try again otherwise we can not connect to the instance. (did t
he browser process quit?)
ERROR    [agent] ❌ Result failed 2/3 times:
 To start chrome in Debug mode, you need to close all existing Chrome instances and try again otherwise
 we can not connect to the instance.
INFO     [agent] 📍 Step 1
INFO     [browser] 🌐  Storing Browser Profile user data dir in: C:\Users\kares\AppData\Local\Google\Ch
rome\User Data
ERROR    [browser] ❌  Failed to start a new Chrome instance: BrowserType.connect_over_cdp: connect ECO
NNREFUSED ::1:9242
Call log:
  - <ws preparing> retrieving websocket url from http://localhost:9242

ERROR    [browser] Failed to initialize Playwright browser: To start chrome in Debug mode, you need to
close all existing Chrome instances and try again otherwise we can not connect to the instance.
ERROR    [browser] ❌  Failed to create new browser session: To start chrome in Debug mode, you need to
 close all existing Chrome instances and try again otherwise we can not connect to the instance. (did t
he browser process quit?)
WARNING  [browser] ⚠️  Page load failed, continuing...
INFO     [browser] 🌐  Storing Browser Profile user data dir in: C:\Users\kares\AppData\Local\Google\Ch
```

Failed to use my own browser

Error fixing



Debugging Chrome

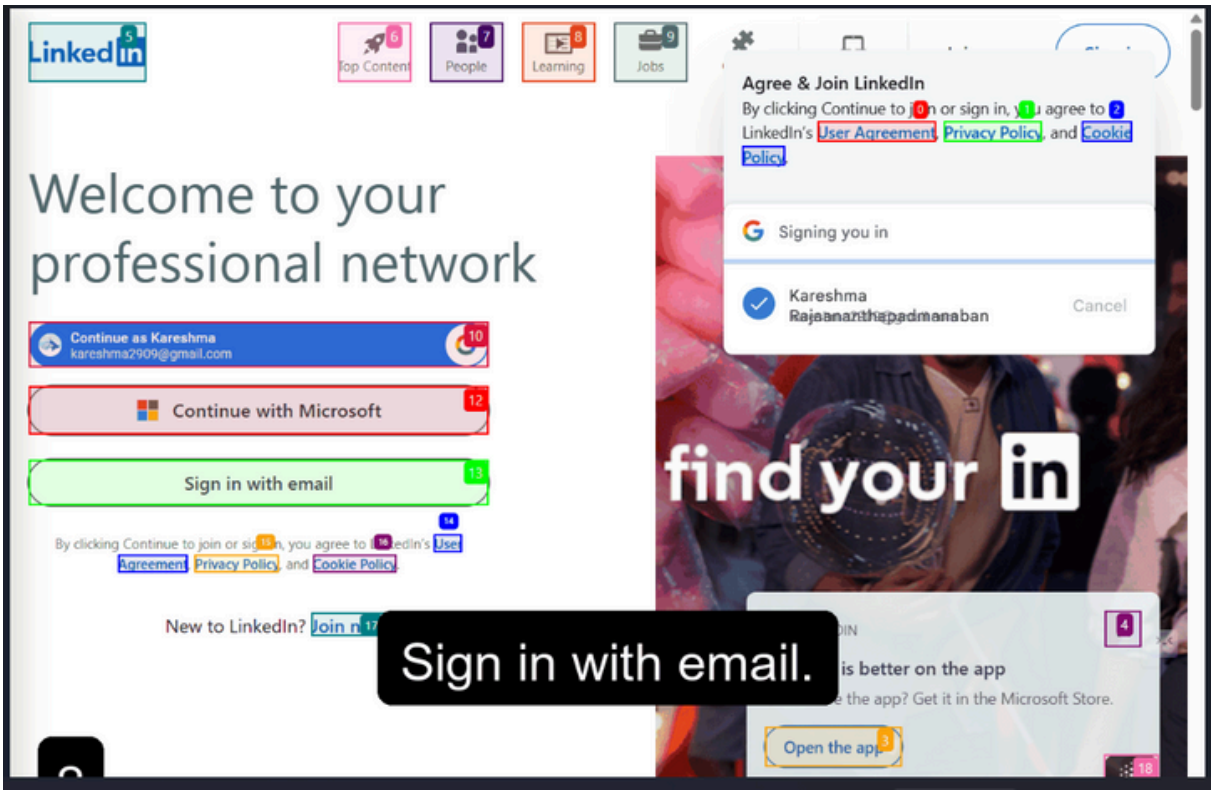
To fix this, I shut down all open Chrome windows and launched Chrome manually using a custom command:
"C:\Program Files\Google\Chrome\Application\chrome.exe" --remote-debugging-port=9242 --user-data-dir="C:\chrome-debug"

This command starts Chrome with **remote debugging** enabled and isolates it using a dedicated user data directory. That allowed the agent to **hook** into my **authenticated browser** session cleanly, without interference from other running Chrome processes.

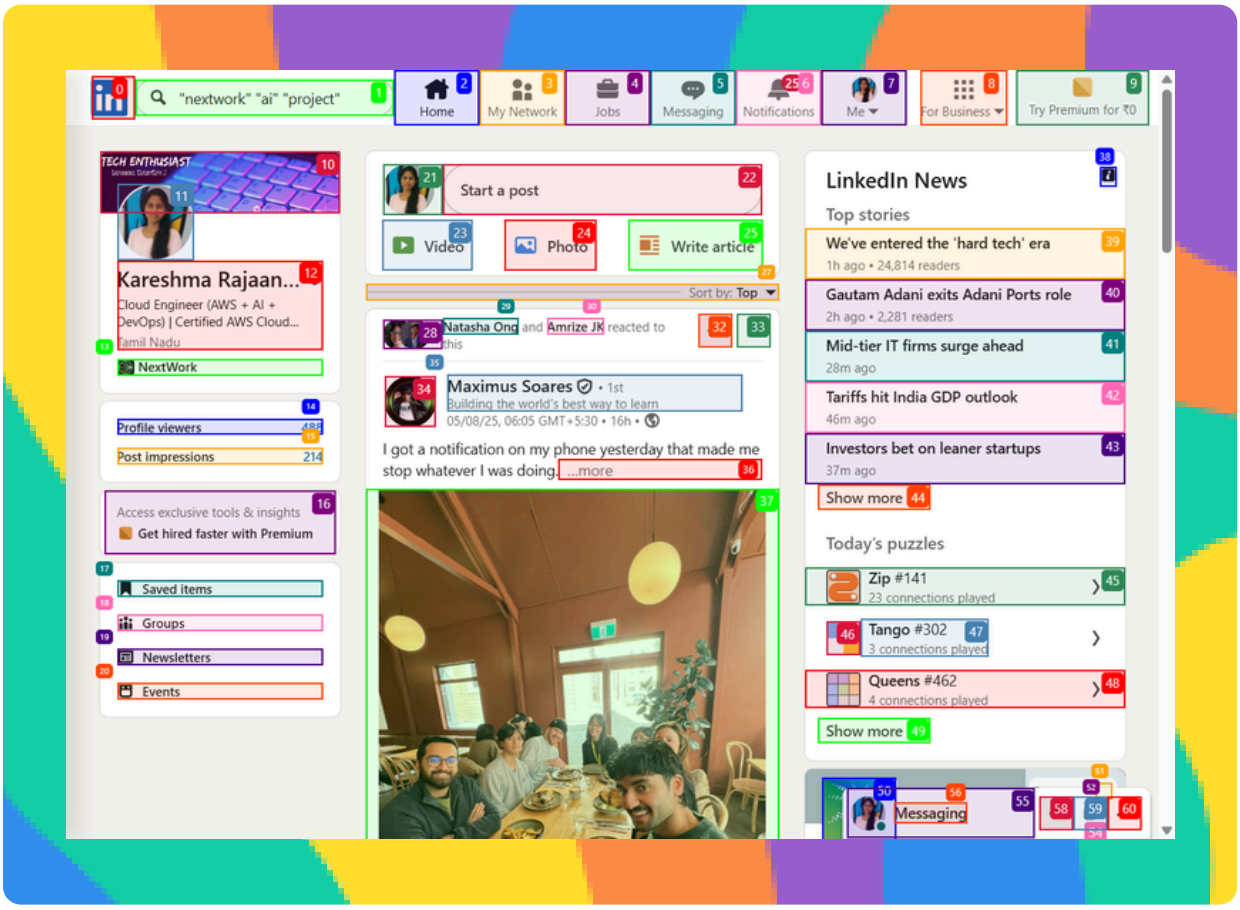
The root cause here was Chrome's **default behavior** of maintaining background processes, which **blocks** external control unless a fresh, debug-enabled instance is explicitly started. Once I did this, the agent worked as expected and was **able** to perform the task in my **personal Chrome browser.**

Kareshma
Rajaananthapadmanaban

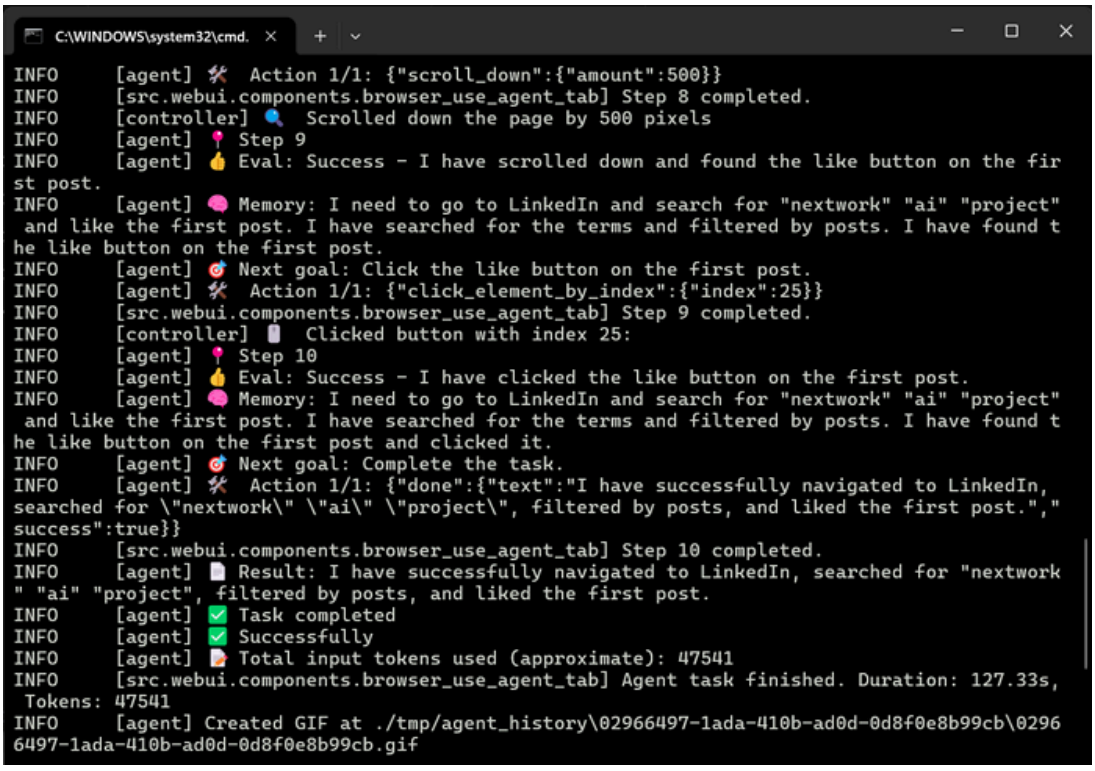
Error fixing



Agent sigining in



Agent controlling my LinkedIn



Logs in the terminal

Kareshma
Rajaananthapadmanaban

Follow up for more interesting projects !!!

Check out nextwork.org / my profile for more projects