BrowseLLM: An Autonomous Agent for Web Interaction

Mehrdad Khanzadeh
mkhanzadeh@umass.edu

Shreyans Babel
sbabel@umass.edu

Rishab Mehta
ramehta@umass.edu

Jayesh Ramana
jramana@umass.edu

1 Problem statement

The world where we have the ability to have consistent and a robust tool for browser automation offers us great benefits. There are endless use cases for which a tool like this has huge utility including automating redundant and boring tasks or tasks like handling job applications. While the already available solutions promise these capabilities, they often rely on a headless browser which prevents day-to-day use cases and does not have a built-in Captcha solver. To address these limitations, our group presents BrowseLLM - a browser extension that can receive tasks from the user, perform the actions on the webpages to carry out the task, answer questions from the webpage, and solve reCaptcha challenges with a click.

BrowseLLM supports all models provided through OpenAI, Anthropic, and Groq APIs. Our goal was to develop and explore a Chrome extension that runs an LLM-based web agent capable of comprehending and interacting with the browser, with an inbuilt Captcha solver.

There are multiple motivations that guide our solution to the problem. The key motivation is to have an agent that can: (1) Receive and analyze tasks from the user, (2) Understand the current browser environment, (3) Devise a strategy for an execution plan, (4) Execute the plan, and (5) Receive feedback from the user. Additionally, this project also aims to solve bot detection security measures and try to remain undetected. We believe that this agent provides us with not only a practical tool to automate repetitive tasks but also multiple technical insights such as understanding LLM behavior with different prompts and context length constraints, evaluating LLM capabilities in understanding the web content and generating relevant code and testing state-of-the-art GPT-V capabilities.

The code and the extension are provided via email. We will discuss implementation details in the upcoming sections.

2 What you proposed vs. what you accomplished

In our project proposal, we planned to integrate several features into our web agent to test how seamless we could make it for users. These features included environment analysis using GPT Vision and/or webpage DOM elements, task decomposition into sub-goals, tackling security challenges like reCaptcha, retrieving relevant information such as codebase documentation, code generation and execution, and post-execution analysis.

During the project, based on our experiments and the constraints, we made some adjustments to better suit the overall scope and time constraints.

- ~~Developed the core system capable of analyzing web environments and generating steps of decomposed actions to perform an overall task~~
- ~~Performed thorough experimentation and analysis of pros and cons of using GPT vision or screenshots of a webpage in our final project~~: Not integrated with the Chrome extension for this version delivery.
- ~~Analyzed and implemented methods for the agent to bypass security measures like reCaptcha~~
- ~~Detailed prompt engineering based on current research and class teachings~~.
- *RAG*: We did not expand upon this based on the feedback on the proposal from the TA and due to time constraints, but we added a feature to the extension where it can answer questions based on the current webpage.

We go into more detail about all the components above in section 6.

3 Related work

The evolution of AI agents has progressed rapidly over the years. Initially, agents were based on symbolic AI, relying on logical reasoning and symbolic knowledge to mimic human thought. As computing capabilities and data sources grew, agents using reinforcement and transfer learning emerged, learning from environmental interactions and complex policies. Most recently, the rise of LLMs has led to LLM-based agents, which exhibit emergent and multimodal capabilities, combining the reasoning strengths of previous generations with enhanced generative abilities. (Xi et al., 2023).

For this project, it is very important for us to have good prompting strategies and there are different approaches researchers have taken for similar domains of tasks. LLMs can be inconsistent and susceptible to reasoning loops which is a major challenge when trying to make a reliable tool (Muthusamy et al., 2023). We utilized chain-of-thought prompting to combat the reliability and hallucination issues. In addition, we experimented with a prompting strategy called ASH (Actor-Summarizer-Hierarchical) prompting (Sridhar et al., 2023). ASH used a summarizer that takes raw observation of the environment and produces features that better suit the task and an actor that takes in these features with all the already taken actions and is responsible for predicting the next action. The authors test this strategy for web navigation and it outperforms ReAct (Reason + Action) (Yao et al., 2023) prompting.

Another important aspect of our project is DOM compression to avoid LLM context length constraints and prevent hallucinations due to large prompts. (Nass et al., 2023) introduces VON Similo LLM, a web elements localization approach that utilizes a multi-locator strategy to find and rank the top 10 candidates based on the task. It then uses GPT-4 to identify the most relevant tags or elements for the task. Consequently, instead of sending the entire webpage DOM in the prompt, only the necessary parts are used. In our approach, we provide the LLM with only the interactable components of the DOM and convert the page content to plain text.

Code generation and execution is a major application of LLM-based agents. There are many methods that explore this such as Wang et al. (2024)'s framework for executable code generation by LLMs which integrates a Python interpreter to dynamically adjust actions based on environmental feedback. Also, Zheng et al. (2024) develop an open-source system that integrates code generation, execution, and iterative refinement using both execution and human feedback. For our Chrome extension, we attempted to generate code through LLM APIs, execute the generated code in the browser, and evaluate the result but pivoted to another method to carry out actions.

Evaluating an LLM agent's attempt to complete a task is crucial to improving the model's accuracy and efficiency. Using another LLM to do this has been explored by Zhang et al. (2023) who use GPT-4V to evaluate multimodal tasks and find that the model is able to achieve a high agreement rate with humans. To check if our agent successfully solves Capcthas, we use GPT-4V to analyze visual elements for confirmation in the webpage before and after the agent executes its actions. Deng et al. (2024) recently introduced a framework for solving Captchas with LLMs using a Domain Specific Language (DSL) (Mernik et al., 2005) that guides LLMs in generating actionable sub-steps for each Captcha challenge.

4 Your dataset

For this project, the primary data that our team used was the Document Object Model (DOM) of the web pages that our agent interacts with. The DOM contains the content and the tree-like HTML structure of a web page, including many visible and non-visible elements and attributes. Whenever a task is given to the agent, its first job is to extract the DOM content of the current web page. This serves as the raw data, which explains the current web page to the model.

DOM consists of HTML elements, also known as tags, that have some attributes, inner children (consisting of other elements or text), and separately attached styles and scripts. This simple structure contains information ranging from how elements should look and behave, to structure, to hidden properties for accessibility readers, and many more. Here is a simple example of an element in a web page:

```
<div class="btn" onclick="alert('Hi!')">
    Click me!
</div>
```

These DOM objects can be huge in size and can contain a lot of unnecessary information irrelevant to the task. One of the main differences between a human using a web page and a bot reading the DOM is that humans are not involved in many details while manually operating

a website, like the URL behind a clickable link. This is why we experimented with various preprocessing and filtering techniques to retrieve the most relevant DOM parts needed for understanding and performing the task. We go into more detail on what DOM compression methods we used in BrowseLLM.

Another piece of data we experimented with was the screenshots of the web page. This was used in combination with the DOM structures to add an extra element of understanding of the current environment for the agent. However, after a few experiments, we decided to only integrate the DOM filtering into the agent for now. The main reason behind our decision was the challenges faced by the LLM to perform the action using only the image. We found that with effective DOM compression, we will be able to significantly reduce the size of the input while having an accurate representation of the webpage.

Below we review our different data preprocessing methods.

4.1 Data Preprocessing

As our extension is based on DOM, the main preprocessing used in our extension is DOM preprocessing. However, as we extensively explored the vision-based approach, you can also find information about data preprocessing while using vision in the coming sections.

4.1.1 DOM Preprocessing

To develop the Google Chrome extension that leverages an LLM to carry out tasks on web pages, we started by extracting the entire HTML source code and feeding it to the model, asking it to generate the JavaScript code to perform the given task, and injecting and executing the code within the page.

This proved to be too many tokens for the LLM to process in each prompt, especially for the more complex websites, so another method to supply the LLM with the information it needed to carry out the user's request was needed. We experimented with using page screenshots for input but ultimately opted to refine the text-based method for our extension since the model was not able to accurately identify the elements on the website and extract the corresponding HTML components required for writing operational JavaScript code using images.

To reduce the amount of source code we gave the model, we started by removing unnecessary tags and attributes, inputting only content tags (without style, script, etc.), and simplifying the DOM by removing the attributes that did not have an impact on model's understanding of the environment.

In our tests with this approach, we found the model to be faulty in properly carrying out the action, potentially due to the removed information and the amount of unrelated information still present. More on that in section 7.

In our next preprocessing effort, we decided to extract clickable elements, input elements, and text content of the page separately. For extracting clickable elements, we implemented several different strategies to identify the clickable elements, then simplified their inner HTML structure by removing their children elements and style and script tags and only keeping their inner text. We also extracted all input fields from the DOM based on their tag name combined with some of their attributes. Both clickable and input elements were further trimmed to only include the attributes that are somehow user-facing, the ones that make humans understand their purpose. The entire page content was also converted to text using a 3rd party library and was also added to the prompt as plain text, stripped of all links and images to prevent any distractions or confusion for the LLM.

Finally, after observing the model's inaccuracies in generating the correct element selector, we added a special attribute (called `llm-index`) to all HTML elements before extracting the elements to enhance the LLM's performance in generating appropriate HTML selectors. Through this process, we refined the input to the LLM, ensuring that the extension could effectively and efficiently automate web interactions by generating accurate and relevant JavaScript commands.

We supplied the LLM with these three components (clickable elements, input fields, and textual content) and asked it to output a JSON object with either an output type or an action type, with the action described as a list of click or insert operations.

4.1.2 Vision Approach

There are two pieces of data being fed into the large language model per interaction in our vision-based approach. One is the user's request, e.g. login, pass the Captcha, etc. The other is an image of the page that the user is currently seeing, taken as a screenshot. The image only focuses on the page itself, so it does not include the URL bar, tab headers, or the dock/-taskbar at the bottom.

Since the vision approach was not integrated with the Chrome extension, it had to be run locally. This necessitated encoding the image in base 64 before making an API request.

The first step was to see if any further preprocessing was required, i.e. would the LLM be capable of understanding a screenshot of a webpage, and what interactions would need to be done on the page to complete a task. The LLM turned out to be very effective at understanding its environment but had some errors/missteps when determining the set of interactions necessary. However, the greatest pitfall came when trying to get the information needed for the interactions. When no other preprocessing is involved the primary way the agent would be able to interact would be through selecting HTML elements based on their coordinates. Unfortunately, as the documentation states, the LLM is "not yet optimized to answer detailed questions about the location of certain objects in an image." (OpenAI, 2024). This statement was backed up by our testing, which means preprocessing would be required.

The first attempt was to retrieve the webpage's viewport size through JavaScript code. This would have the window's height and width in pixels. The idea was that including this information in the prompt would allow the LLM to use it as a reference when determining location. This method was abysmal, it turned out to be equally as bad as no preprocessing at all.

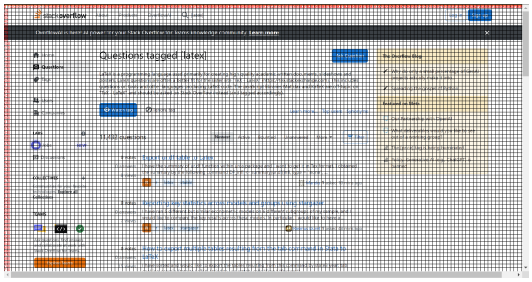

Figure 1: Grid line preprocessing with 10px distance between lines and axis labels shown. Underlying webpage: https://stackoverflow.com/questions/tagged/latex

The next attempt still followed along the lines of directly determining the pixel coordinates of elements. This approach involved programmatically overlaying a grid on the webpage, with each pair of lines having 10 pixels between them. An example of this is in Figure 1. Theoretically, the LLM could either use these for reference or try to count the number of lines to determine the coordinates. This was tested with varying distances between gridlines (10, 25, and 50), as well as showing or hiding pixel distance axis labels. These all suffered from the same lack of success. Furthermore, even if this attempt had succeeded, the element would most likely need to be on a gridline intersection in order to have accurate coordinates.

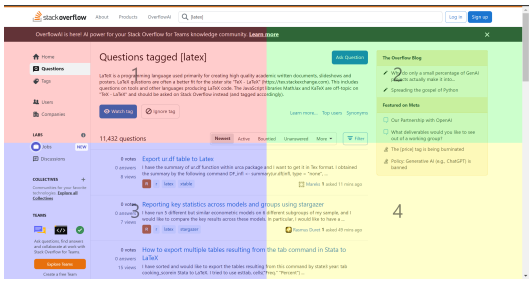

Figure 2: Colored quadrant preprocessing. This example has numbered quadrants. Underlying webpage: https://stackoverflow.com/questions/tagged/latex

Another preprocessing attempt was to iteratively hone in on an element. This is shown in Figure 2, and involves splitting the page into four quadrants with JavaScript injection via the console. The basic idea here is to have the LLM determine which quadrant the element it is attempting to locate lies in. Based on that decision it will be fed a new image of just that sector, once again split into these four quadrants. That way it would be able to narrow down its focus until it is only looking at the element, and the coordinates could be calculated based on the chain of decisions. This was attempted with and without the number overlays seen in Figure 2. This was significantly more successful than the grid approach. However, the downfall was due to the long chain of continuous requests. If any one of them returned the wrong quadrant it would be impossible for any of the following requests to ever find the element. This made it a lot more error-prone than it would otherwise be. This method was also much more expensive, as it required many LLM calls. Finally, if an element was on a borderline between quadrants it would become difficult to focus on it.

This approach switches focus from trying to determine the coordinates of an element to identifying interactable elements and then selecting an element based on its unique tag. Initially, this was done through the Chrome extension Vimium. Crosby and Sukhar (2009). One of the functionalities of this extension is displaying tags near buttons and input fields that would allow a user's cursor focus to transfer to that el-

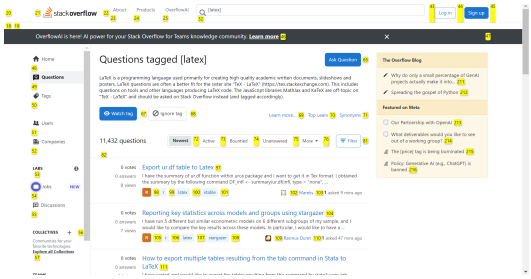

Figure 3: Index tagged preprocessing. Underlying webpage: https://stackoverflow.com/questions/tagged/latex

ement via generated links. It is similar to Figure 3, but it would have a letter sequence in its tags instead of an index number. For instance "SS" could be the tag next to an input field. This approach worked very well for identifying elements. The LLM was already able to determine what element was needed for its task, and it is now capable of recognizing the characters associated with the tag for that element. The problem here is with the Vimium extension itself, as there is no easy way to programmatically interact with it.

Because of this blocker, we decided to recreate Vimium tags on our own, and this also gave us the freedom to add some more information. Using JavaScript injection via the console we filtered for links, buttons, input areas, focusable elements, and a few other possibilities. This is a similar process to the filter used by the Chrome extension. Each of these elements was given a new field with consecutively increasing index values, and these same values were used in their tags. This allows us to directly reference an element with JavaScript based on its tag number, which the LLM can identify. Furthermore, due to this being our own implementation, it allows us to get around Vimium being blocked on certain sites, such as Gmail. An example of this is in Figure 3. This method of preprocessing worked the best by far. Since this was something we implemented ourselves there are definitely still improvements that can be made. On some pages, there are fields that do not get picked up by our filter. Occasionally there are also problems with tag placement, where certain tags will overlap or shift a little too far from their parent element. These are both issues that can be resolved with some more development time, and the most important part, which is having a way for the LLM to interact with elements, works. Thus, this was the preprocessing method that was chosen for our vision-based approach.

4.1.3 Captcha

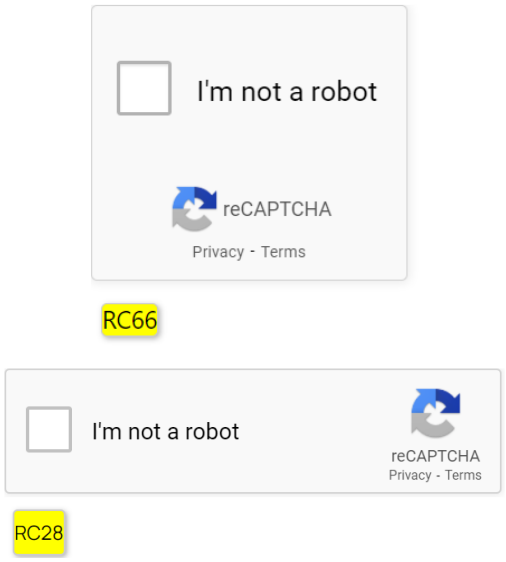

Figure 4: Captcha preprocessing with specialized tag. Underlying webpage: https://stackoverflow.com/nocaptcha?s=806507fb-0ed9-4dbf-8bef-d49cf6e7f643 and https://2captcha.com/demo/recaptcha-v2

For Captcha-specific preprocessing with the vision-based approach, we need to alter our filter and tagging methods. The filters were expanded to include common Captchas, such as eeCaptcha. We also changed the tags, so when something such as a reCaptcha checkbox comes up the indices are prepended by "RC" as you can see in Figure 4. This tagging method does fail on Google's reCaptcha demo due to their "Content Security Policy."

4.2 Data annotation

Our project does not involve any annotation. Future work may include annotation of test data sets. For the vision method, these would involve an image of a website and a task pairing, as well as a list of actions that a human would find optimal to complete the given task. For the DOM-based method it would need the same pieces of data, but with the DOM instead of an image. This data set could be used for evaluation as well as fine-tuning.

5 Baselines

To comprehensively evaluate the capabilities of this project, we conducted multiple experiments that served as our baseline. All baselines were chosen because they were the most naive way to solve a task in one iteration. Having these as baselines would allow us to compare what,

if any, improvements our approach made. We separated ourroject into 2 main components :

5.1 Chrome extension using only the DOM

For baseline (a) we used a zero-shot prompt approach by feeding the entire DOM to the LLM to perform a task in 1 iteration.

5.2 Using GPT vision

Similarly to the chrome extension baseline, we used the image of the environment without preprocessing to the LLM to perform a task in 1 iteration. We had two kinds of baseline experiments for vision. [1]

5.2.1 Interaction

This baseline focused on the ability to effectively interact with the webpage. The LLM was given an image of a webpage and asked to complete a task that would not require it to leave that page. In other words, it would not be required to understand the information on other pages connected to this one. Specifically, the request made to the LLM was to generate JavaScript code per action that could be injected to complete the task. The goal of this baseline was to show the LLM's existing capabilities as far as executing actions on a page go. Some examples of tasks in this baseline are "Click the cart button" from the Amazon home page and "go to the politics category" from CNN's homepage. For this baseline we only collected data on whether the task was successfully completed or not.

We had 50 experiments for interaction. An experiment was considered successful if the JavaScript code generated by the LLM accomplished the objective. 34% of the experiments were successful for this baseline following this definition. This poor score was largely due to the fact that the baseline was unable to pass any of the eight experiments we had that included Captchas. The agent was flagged for non-human activity and blocked from solving the Captcha.

5.2.2 Multi-Page

In this baseline, the LLM was once again given an image of its starting webpage and a task. This task involved navigating and taking action on multiple pages. For instance, from the Amazon home page "Search for "Harry Potter book series", and add the boxed set to your cart." Here we were testing for the action list, to see if the LLM was able to choose the correct actions. It was successful with the Harry Potter task. We did not require it to generate any code, just return a list of "click", "input" and "read" actions describing what steps it would take. The goal here was to see how well the LLM could plan its actions.

We had 30 experiments for multi-page. An experiment was considered successful if by following the steps generated the task would be completed. Using this metric, 40% of the experiments were successful for this baseline. We also gathered a few other data points about whether the baseline created extra steps, missed steps, or had ambiguous steps. For reference, the baseline list had about 3 steps on average. The generated actions were compared to an ideal action list that was manually created per task. There were 0.4 extra steps on average, 1.23 missing steps, and 0.43 ambiguous steps. We used our judgment to determine when a step should be considered ambiguous. For example, one task was starting at the YouTube homepage and "go to first subscription's community tab and like the first post." An action that the baseline created was "clicking on the first channel icon in the subscriptions list". Due to its lack of knowledge, the LLM made a vague statement instead of specifying what channel should be selected.

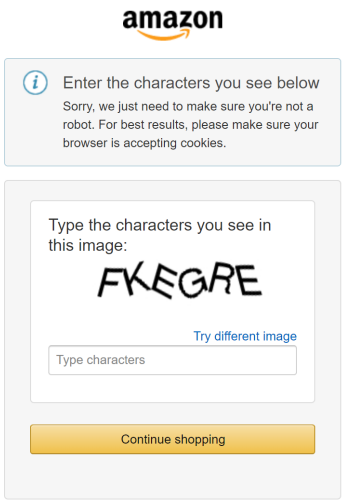

Figure 5: Captcha task. Underlying webpage: https://www.amazon.com/

5.2.3 Captcha

The LLM was asked to solve Captcha's as part of the previous two baseline experiments. For instance, the LLM was asked to pass the Captcha in Figure 5 as part of one of the multi-page tasks. There were 5 Captcha-related tasks in

[1] Some examples of the experiments can be found here are shown in the Github repo

the multi-page baseline, and it passed 2 of them. There were 8 interaction experiments involving Captcha, and the baseline passed none of them.

6 Your approach

The approach we used in the extension breaks down as follows:

- After providing and saving API access tokens, users input a task inside the extension's input.
- An injected script will extract and filter DOM, as described in the data processing section, and will return a JSON object containing the clickable elements, input elements, and content text.
- The following prompt is made to the selected LLM:

```
`You will receive two inputs. The
↪  first one is a task (starting
↪  with "Task: "), and the second
↪  is a JSON object, with the list
↪  of all the inputs, clickable
↪  elements, and the content in the
↪  DOM (starting with "DOM: ").
Your ultimate goal is to solve the
↪  task. The task either requires
↪  you to output something or
↪  perform some actions on the
↪  webpage.

If you think an output is requested,
↪  for instance, the summary of the
↪  page, or for answering some
↪  questions, you should return a
↪  JSON in this format:
{
  "type": "output",
  "content": <content>
}
If you think you need to perform an
↪  action, for instance, filling
↪  out an input field, or clicking
↪  on a button or link, you should
↪  return a JSON in this format:
{
  "type": "action",
  "content: [
    {
      "type": "click",
      "llmIndex": <The 'llm-index'
      ↪  attribute of the clickable
      ↪  element that should be
      ↪  clicked in the page>
    },
    {
      "type": "input",
      "text": <Text to be put inside
      ↪  the input>,
      "llmIndex": <The 'llm-index'
      ↪  attribute of the input
      ↪  element that should be
      ↪  filled with the text in
      ↪  the page>
    },
    ...
  ]
}
You have as many actions is one step
↪  as you want, as long as you can
↪  be sure it's the correct action.
↪  Actions are either "click" or
↪  "input", are will be run in
↪  order you provide.
Make sure you return the result in
↪  the required format. You should
↪  only return a JSON object,
↪  nothing else.

Task: ${prompt}

DOM: ${dom}`
```

- Based on the output "type," the extension either shows the message to the user or iterates through the list of actions and executes them one by one using the functions we implemented in the extension that can either perform click or type some input into the page.

Our implementation, made possible after multiple rounds of iteration and improvements, proved to be effective for solving many zero-shot tasks on a web page.

In the subsections below, you can read more about the details of different experiments.

6.1 Prompt Engineering

An important focus of our project was on experimenting with multiple prompting strategies to get the maximum out of the LLMs' ability to analyze environments, decompose tasks, and generate code.

- DOM Prompting: One straightforward approach we used was directly feeding the whole DOM representation with the task and asking it to generate a set of actions to perform on tags on the web page.
- Multi-modal Prompting: Going beyond the just text, we also incorporated images of the environments but after a few experiments we decided that it is better to focus on just improving the DOM-based approach.
- Zero/One shot and CoT prompting: We also experimented extensively with different prompting techniques taught in the

class. For one-shot prompting experiments, an example input-output pair was added to the prompt, and the pair contained explicit multi-step reasoning so that the LLM could use that as a reference for desired output.

6.2 Vision

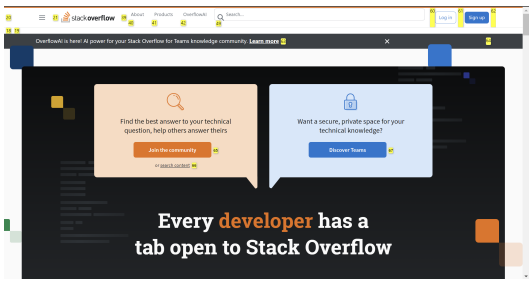

Figure 6: Preprocessing on stack overflow homepage. Underlying webpage: https://stackoverflow.com/

For our vision-based approach, we used GPT-4V exclusively, up until the release of GPT-4o, at which point we switched to that due to it being more cost-effective.

The approach can be broken down into five major steps. There will also be an example of the output from each step for the task "go to the login page" and the image in Figure 6.

1. **Initial Action List:**
   In this step, the LLM is sent a preprocessed image of the current webpage as well as the user's task prompt. Similarly to the way the Chrome extension is set up, the LLM is initially asked to determine the type of task. This could either be something related to what the page is about, such as providing a summary or answering questions, or it could be needing to execute actions. If it is the first case then the LLM is asked to return a JSON object that states the task was of type output, and it will also return the content that was requested. In the latter case the LLM is asked to return a JSON object that states the task was of type action and a list of the actions that would need to be taken to complete the task. Each action item contained a type such as click or input, and a description of the field that would need to be accessed. Input actions also had an additional field for the text that would be inserted. An action that this approach is currently lacking is the ability to scroll. We were unable to develop a good way to prompt and integrate this action in the time we had, but it is an important consideration in future work. This is done through zero-shot prompting and the GPT API's enforced JSON response.

   - If it was the first case then the flow will end here, and it will output the summary or text that was requested.

   Here is an example of the generated action list for solving a Captcha.

```
{
  "type": "action",
  "content": [
    {
      "type": "click",
      "field": "Log in button"
    }
  ]
}
```

2. **Find Elements:**
   In this step, the LLM is given the action list generated by the previous API request and the preprocessed image of the webpage again. The action list contains the type (click or input) and the description of the field that was generated. It is tasked with finding the tagged index number that corresponds to each element in the page that is needed for an action. The goal is for it to return a list of numbers, where each one corresponds to its respective action's element. The action list from the previous step needs a slight bit of processing in order to correctly format it for this step. The request for finding the elements is done with chain of thought prompting. The LLM is asked to apply each step per action it needs to take. Here is a brief description of the steps:

   (a) Understand that the goal is to find the number corresponding to a field.
   (b) Locate the field needed for the action.
   (c) Identify the numbers.
   (d) Verify that these are the correct numbers.

   Here is an example of the numbers found to correspond to each action. We can see that these are correct according to Figure 6.

```
{
  "actions": [
    61
  ]
}
```

3. **Interact:**
   In this step, the LLM would use the action and element ID pairings to interact with the webpage. The JavaScript functions for each type of interaction (click and input) have been previously generated. This could be done with GPT's function calling capability, in which we would provide a description of the generated functions and it returns JSON that can then be used to call the functions. It may also be possible for this to work directly with a script that maps the action type to the correct function and determines the function parameters. Unfortunately, we did not have time to implement this so are manually executing the functions based on the LLM responses from the previous two steps. Here is an example of the function calls required to execute the two actions that have been generated.

```
lst = visibleFocusableElements
clickElementAtIndex(lst, 61)
```

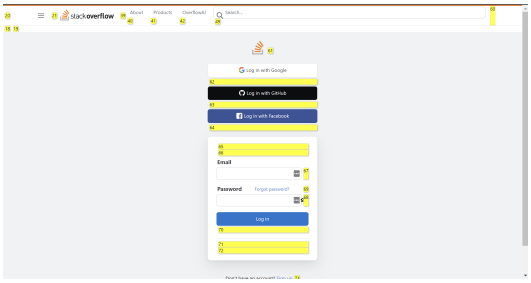

Figure 7: Stack overflow login page. Underlying webpage: https://stackoverflow.com/users/login?ssrc=head&returnurl=https%3a%2f%2fstackoverflow.com%2f

4. **Determine Further Actions:**
   Here, we give the LLM three pieces of information. The original task that the user input, the list of actions that were taken, and a preprocessed image of the webpage after those actions were completed. The LLM is asked if further actions need to be taken to complete the task. It is meant to return JSON, with a boolean that is true when more actions are required and a sentence for its reasoning for why the task is completed or not. This sentence is there for us to understand where the LLM may be failing, and would not be shown to the user. If the LLM claims that more actions are needed then the flow goes back to a slightly altered version of step 1, which states that some actions may have been completed already, and restarts. This altered version also includes the list of actions up to this point. This is done with Chain of Thought prompting. The LLM is given steps to help it make its decision, here is a brief description of each step:

   (a) Consider what a webpage when the task is completed would look like.
   (b) Compare that to the webpage image provided.
   (c) Consider whether the steps taken would allow you to complete the task.
   (d) Use the information from the previous three steps to make the decision.

   Here is an example of the LLM determining whether additional actions are required. The new image provided for this request is in Figure 7.

```
{
  "more_actions": 0,
  "reasoning": "The login page is
      visible, and the task to go to
      the login page is completed."
}
```

5. **Evaluation:**
   Finally, we need to determine if this was successful. This can, and should, be done at least partially through manual grading of success. To supplement that, and provide an automated alternative, we asked the LLM to evaluate itself. This is done by providing it with three pieces of information. The original task prompt, an image of the starting webpage, and an image of the webpage after all actions have been carried out. It is then asked to determine if a task has been successfully carried out. It will return JSON with a boolean that is true for successful completion as well as reasoning for the decision it made. Here is an example of the LLM evaluating whether the task was successfully completed.

```
{
  "completed": 1,
  "reasoning": "The second image
      shows the login page, which
      means the task to go to the
      login page was successfully
      completed."
}
```

There was a lack of one-shot and few-shot prompting in this approach. This is due to not wanting to overload the LLM with too many tokens from images being used as examples.

The only non-standard library used for this approach is OpenAI, which was necessary to make requests to GPT-4V and GPT-4o. All testing for this approach was done with Windows x64 and the Chrome browser. Changing the local machine would not have any effect. As previously mentioned we were unable to properly implement Step 3 in time, and this approach was also not integrated with the Chrome extension.

Similarly to the baseline, we split the experiments into interaction and multi-page tasks. There were 30 multi-page tasks and 50 interaction tasks. The number of experiments was restricted due to the flow not being fully automated, which made them much more time-consuming. We collected the same metrics described in the baseline section.

- Multi-Page: 76% of these experiments were successful with our approach. The approach needed to make 4.4 calls to GPT per task on average. One call could either be trying to generate a list of actions or determining whether or not the task has been accomplished. As a reference for the following statistics, the approach created about 6 steps per task on average. There were around 0.39 extra steps on average, 0.51 missing steps, and about 0.09 ambiguous steps on average. We can see that this had more steps than the baseline action list. This is due to the baseline only seeing the initial webpage, while the vision approach sees the updated webpage after each set of actions. Both baseline and our approach had about the same number of extra steps, but the vision approach had significantly fewer missing steps—0.51 to 1.23. Finally, there were much fewer ambiguous steps. Less than 0.1 for our approach as compared to the 0.43 of the baseline.

- Interaction: 80% of these experiments were successful with our approach. There were about 3 calls to GPT per task for these baselines. The drop in relation to the other set of baseline experiments is because it was much rarer for step 4 of our approach to determine that the task was not finished. This is expected since all the tasks in this baseline are supposed to be achievable in a single page. In other words, there was no need to process the information from a second page and take more action. Our approach did much better than the baseline in which only 34% of experiments from this set were succesful. As we previously mentioned in the baseline section, there were 8 Captcha specific tasks in this set. Out of those our approach was unable to pass any of them. This is due to the generated JavaScript functions being used to interact with the page. Even though our approach was able to understand what the Captcha was looking for, for example, the word in Figure 5 and where it would need to click, it was unable to execute those actions. As we said in the baseline, we believe this is because the sites are detecting that we are programmatically manipulating elements which is blocked. In the future, we would switch how the Vision approach interacts with the webpage to be similar to the Chrome extension, as that had more success.

6.3 ReCaptcha Solver

A key goal listed in the proposal was the agent's ability to solve Captchas autonomously while remaining undetected as a bot by security measurements. There were multiple experiments that we conducted on multiple different types of Captchas, and we were able to fully tackle reCaptcha.

Here we discuss our audio-based approach to reCaptcha, followed by our vision-based attempts.

6.3.1 Audio-based approach

After experimenting with reCaptcha, we noticed an accessibility feature built into reCaptcha that was much easier than solving multiple visual puzzles: an audio challenge that should be transcribed.

By manually inputting the audio challenge into Speech-to-Text services, our initial tests proved that the audio challenge can be solved automatically.

Upon further research, we came upon some previous work that tried this approach before (Bock et al. (2017), thicccat688 (2023), etc.), but some were old projects for older versions of reCaptcha, and some were not packaged as a browser extension, relying on running a headless browser or Python script to achieve it, or needed manual human intervention for some parts.

We added a button to our extension that automatically solves reCaptcha with one click, using

OpenAI's Whisper (Radford et al., 2023) API. We found that this model is pretty robust in detecting the audio.

The task involved injecting a communication channel and control code in the reCaptcha iframe, clicking on some buttons in order to activate the audio challenge, extracting the audio challenge URL from DOM, sending it to the extension to send it to OpenAI's Whisper for detection, and injecting back the response in reCaptcha. The main challenge in developing this functionality was to behave as such to remain undetected as a bot, as perceived by Google. Our initial attempts to automate the clicks were blocked by Google, outputting the bot detection error. It was getting triggered by programmatically clicking on the audio button.

After some experiments, we created a custom clicking mechanism that better mimicked human behavior, by introducing some delay between `mousedown` and `mouseup` events, and better simulating click properties like event bubbling. We further improved it and made it more robust by introducing random delays instead of fixed ones and also adding a delay between each step of solving the Captcha. All the added delays made our Captcha solver slower, but it can solve reCaptcha within some seconds with just one click. Here is the breakdown of communication between the extension and the injected code inside reCaptcha.

```
export const solveCaptcha = async () => {
  const delay = (ms: number) => new
  ↪  Promise(resolve => setTimeout(resolve,
  ↪  ms))
  await message('startCaptcha')
  await delay(2000 +
  ↪  Math.floor(Math.random() * 1001))
  await message('startCaptchaAudio')
  await delay(1000 +
  ↪  Math.floor(Math.random() * 1001))
  const url = await
  ↪  message('getCaptchaAudioUrl')
  await delay(1000 +
  ↪  Math.floor(Math.random() * 1001))
  if (url) {
    const text = await speachToText(url)
    await message('setCaptchaText', text)
  }
}
```

Here is the click function we used instead of the native `.click()` function of the elements.

```
const delay = (ms: number) => new
↪  Promise(resolve => setTimeout(resolve,
↪  ms))

const click = async (el: HTMLElement) => {
const mouseDownEvent = new
↪  MouseEvent('mousedown', { bubbles: true,
↪  cancelable: true })
const mouseUpEvent = new
↪  MouseEvent('mouseup', { bubbles: true,
↪  cancelable: true })
const clickEvent = new MouseEvent('click', {
↪  bubbles: true, cancelable: true })

el.dispatchEvent(mouseDownEvent)
await delay(200 + Math.floor(Math.random() *
↪  101))
el.dispatchEvent(mouseUpEvent)
await delay(Math.floor(Math.random() * 51))
el.dispatchEvent(clickEvent)
}
```

Utilizing this approach, our extension was able to successfully bypass Google's reCaptcha on various attempts.

6.3.2 Vision-based approach

We also experimented with vision-based approaches which did not end up in our implementation due to various challenges and constraints and overall infeasability. GPT-4V and 4o out of the box were good at solving text-based Captchas that require the user to enter a series of letters or numbers. They would sometimes misinterpret the letters, but due to the vision flow that checked whether a task was complete or not, the agent was able to try again until it succeeded. For the checkbox reCaptchas we attempted to give a specialized tag that would allow the LLM to focus on it. This had some degree of success, but our preprocessing code was not refined enough to accurately target the checkbox. This meant that even though the LLM understood that the checkbox needed to be clicked on, it was unable to carry out the interaction.

One of the recent solutions that aims to bypass Captcha is i-am-a-bot (Ramachandran, 2024). This experiment uses Vertex AI and Gemini-vision to solve Captchas. The flow of the solver agent can be found in Figure (8). It works by checking if the webpage image is a Captcha, selecting whether to use the math or the text LLM-based solver and using that to solve the Captcha. If an unfamiliar Captcha is seen, we either turn to the vision-based approach or handle the error appropriately.

Although it can solve multiple forms of Captcha, it is a Python project that works with images of Captchas and does not provide a medium for carrying out the task.

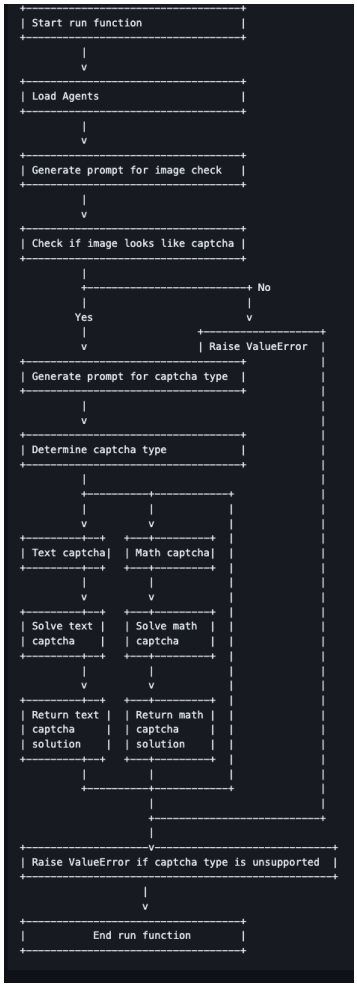

Figure 8: Agent Flow for Captcha Solving. Taken from (Ramachandran, 2024)

7 Error analysis

Our iteration of extension from the baseline involved running multiple rounds of testing and analysis to better understand the limitations and edge cases that needed to be addressed.

The first types of errors we encountered with the baseline were because of the raw size of the DOM. Although working for some small web pages like the main Google search page, it did not work for the majority of pages as it could easily surpass our models' maximum context length of 128k. These errors were eliminated in our final solution, as the input data was trimmed and filtered and in all of our experiments, no web page passed the 128k limit.

Another type of error that prevented the proper execution of code, both in the baseline and our final implementation, was the return format. Although in our original prompt for the baseline, we required the LLM to return only an executable code and nothing else, many models struggled to keep this format and appended the code with descriptions or extra text. In our experiments, we found models to be better at understanding positive tasks, so in our prompt, we asked the model to fence the code using a series of dash characters, and we manually extracted the code using the fence. This method was effective and did not result in further errors in our baseline. Similarly, for our latest model, when we only asked for a JSON object response, it was often polluted with other text. This time, we used JSON's structure to detect and extract it and it fixed the issue.

The other type of error was the JS code generation mistakes by the LLMs. As our baseline directly asked the LLM to generate the required JS to perform the whole task all at once, many different errors occurred. In some instances, the model returned a function to perform the task, which was not immediately executable and prevented the extension from running it. In some other cases, it generated incorrect selectors for the actions it wanted to perform in its code. It came both from a lack of ability to identify the correct element to interact with and sometimes we observed that the LLM finds the element that it needs to interact with and uses the correct selector for that, but also appends the selector with some keywords which made the selector invalid. In very rare cases, it generated irrelevant code.

Because of the errors we observed, we decided to mimic human behavior by only asking for either a click or a text input action and trimming the DOM as much as possible not only for token optimization but to prevent confusing the LLM from detecting the correct actions and elements. We also decided not to rely on the code generation of the LLM, but only ask the LLM to identify elements to be clicked or filled out using a specific index we attached to them and only return the index to us, as previously described. This approach severely reduced the related errors. Still, wrong elements would get selected by the system, but at a much lower rate.

However, trimming introduced another error in our model. In some cases, we trimmed too much information that actually prevented the model from understanding the functionality of an element. We also had some initial flaws in extracting both clickable and input elements. In our manual testing on a multitude of different websites, we found missing (and extra) attributes, tags, and properties in our filters that prevented a specific action from being carried

out on a webpage. We found that information to be general (e.g. a specific type of input we missed) and by adding them our system was robust enough to carry out many tasks.

Another category of errors for both models was because of timing. Some actions needed some delay to be reflected on a webpage, and both the baseline, by immediately executing multiple lines of code, and our main model, which iterated and executed tasks without waiting for the previous ones, could fail on some of those tasks/webpages.

In the upcoming subsections, you can read more about our error analysis for our vision experiments.

7.1 Vision

7.1.1 Interaction: Baseline

This baseline showed that the LLM was often able to come up with the correct series of actions, but its downfall was generating code to execute those actions. For example on the task "click on the Best Sellers button" from the Amazon homepage the LLM generated the code

```
document.querySelector('a[href*=
↪  \"bestsellers\"]')).click();
```

So the LLM understood the task, but due to the lack of information on how the button would be tagged just had to guess that "bestsellers" would be able to find it. This had some success, especially with elements that had straightforward identifiers, but for the most part did not work. We already discussed the LLM's failures with the captcha in this baseline.

7.1.2 Multi-Page: Baseline

There were a few major reasons for failures in this section. The first was lack of ability. For example, one task was "Search for "men's running shoes", filter by brand "Nike", price under $100, and rating 4 stars & up" from the Amazon home page. The action sequence generated was

```
{
  {
  "type": "action",
  "content": [
    {
      "type": "input",
      "text": "men's running shoes",
      "field": "search bar"
    },
    {
      "type": "click",
      "field": "search button"
    },
    {
      "type": "click",
      "field": "brand filter - Nike"
    },
    {
      "type": "click",
      "field": "price filter - under
          $100"
    },
    {
      "type": "click",
      "field": "rating filter - 4 stars
          & up"
    }
  ]
}
}
```

This seems like the correct sequence of actions, and it is for the most part, but it fails at the step where it tries to click a price filter. When doing this search you see that no filter represents under $100, instead, there is a slider for the price range you are looking for. The baseline approach has no way of knowing this as it makes the action list based on the first page it sees, Amazon's home page, in this case. Furthermore, even if it had guessed that there would be a slider, the baseline model is only given the actions of clicking and writing, so it would lack the capability to complete this task.

Another failure point was a lack of foresight. As previously mentioned, the baseline had no way of knowing how exactly the following pages would look. Because of that, sometimes it would ignore any instructions relating to those pages and only generate actions for its current page. This is also a big reason why the number of steps per task is so much higher for the vision approach than the baseline. An example of this is in the task "Go to the shorts tab and read the description of the first video" from the YouTube homepage. The response was:

```
{
  "type": "action",
  "content": [
    {
      "type": "click",
      "field": "Shorts tab"
    }
  ]
}
```

As you can see, the LLM understands that it needs to navigate to the Shorts tab, but just

stops there. This was a recurring theme in many of the experiments.

Finally, the baseline failed because it only had one try. For instance, the task "Pass the Captcha and search for t shirts" which also started on the Amazon homepage. Here the response was:

```
{
  "type": "action",
  "content": [
    {
      "type": "input",
      "text": "ARAWJF",
      "field": "captcha input field"
    },
    {
      "type": "click",
      "field": "Continue shopping button
          "
    }
  ]
}
```

First of all you can see that in this example the LLM also suffers from the previous failure we discussed. In addition to that, it generated the incorrect text to pass the Captcha. In comparison, our vision approach was able to complete this task even though it also made the same error with this Captcha. This is because after doing a set of actions it analyzes the resulting environment to see if more steps need to be taken. This allows it to take multiple attempts at difficult problems, such as a Captcha.

7.1.3 Interaction: Vision Approach

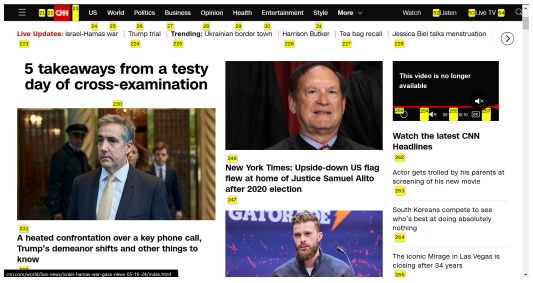

Figure 9: The preprocessed CNN homepage.

The vast majority of failures for this set of experiments came from the Captchas, which we already discussed. The other error was when pages were very complicated. One task from the CNN homepage was "Go to the politics category". Theoretically, this should be simple enough for the agent as it succeeded on other tasks similar to this. Here is the agent's response:

```
{
  "type": "action",
  "content": [
    {
      "type": "click",
      "field": "Politics category"
    }
  ]
}
[{'type': 'click', 'field': 'Politics
    category'}]
Actions:
- click on field 'Politics category'


{
  "actions": [28]
}
```

So the agent understood that it needed to execute a click action, but it ended up choosing the wrong label. As we see in Figure 9 the page is somewhat noisy, especially in the top section. We believe the reason for the failure of this task was due to all the tags nearby in that area.

7.1.4 Multi-Page: Vision Approach

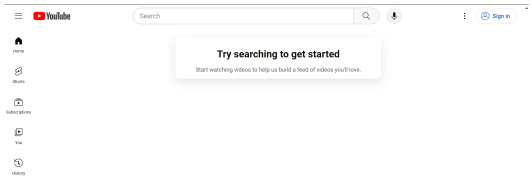

Figure 10: The YouTube homepage.

As we mentioned in section 7.1.2, the vision agent failed on the "Search for "men's running shoes", filter by brand "Nike", price under $100, and rating 4 stars & up" due to the inability to interact with sliders. Another point of failure can be shown through the task "Go to trending tab and tell me what the first video is about" which started on the YouTube homepage. The response was:

```
{
  "type": "action",
  "content": [
    {
      "type": "click",
      "field": "trending"
    }
```

```
  ]
}
{
  "more_actions": "1",
  "reasoning": "The webpage is still on
      the home page and does not display
       the trending tab or any videos."
}
{
  "type": "action",
  "content": [
    {
      "type": "click",
      "field": "trending"
    }
  ]
}
```

As you can see in Figure 10 there is no trending button on the homepage. We cut off the agent's attempt after these actions as it was beginning to loop through the same inputs and responses. Our hypothesis for this failure is that the LLM was trained on a large amount of past data, which includes when YouTube still had the trending tab directly linked from the homepage. So, even though the image is showing something different, it is confident that the trending button exists. In the future, we would need to have it give more weight to the image input. Another issue that this exposed is the potential of getting stuck in an infinite loop. We would need to add a way to check and stop that from happening.

The final error type is forgetting. One task was "Find the 10th best movie and read the description" from the IMDB homepage, and the agent response was:

```
{
  "type": "action",
  "content": [
    {
      "type": "click",
      "field": "Menu"
    },
    {
      "type": "click",
      "field": "Top Rated Movies"
    }
  ]
}
{
  "more_actions": "1",
  "reasoning": "The current webpage only
      shows the menu options; it does
      not display the list of top-rated
      movies or the description of the
      10th best movie."
}
{
  "type": "action",
  "content": [
    {
      "type": "click",
      "field": "Top 250 Movies"
    }
  ]
}
{
  "more_actions": 1,
  "reasoning": "The description of the
      10th best movie has not been
      viewed yet; more actions are
      required to click on the movie and
       read the description."
}
{
  "type": "action",
  "content": [
    {
      "type": "click",
      "field": "The Good, the Bad and
          the Ugly"
    }
  ]
}
{
  "more_actions": "0",
  "reasoning": "The description of the
      10th best movie, 'The Good, the
      Bad and the Ugly,' is visible on
      the webpage."
}
```

Although the agent determined that this task was completed, it had never actually returned the description that was requested by the user. Even though there was no action returning that description, since the request that checks for completion could see it assumed it had already been done. To resolve this we will need to force the LLM to pay more attention to this list of actions, and especially what is missing from them.

8 Contributions of group members

- Mehrdad: Everything related to the Extension - Created the extension from scratch, which included creating the interface for it, token storage, the logic for communicating with the webpages, reading and modifying their content, and injecting code in them,

implemented the security requirements to enable the extension to carry out such access and actions, designed the extension prompts and the task-breakdown structure, ideated, tested, and implemented preprocessing strategies, ideated and implemented the reCaptcha solver in the extension that utilizes audio, did the continuous error analysis for the extension, wrote some of the report sections more directly related to the extension and its methodology, and polished and finalized many other parts of the report.

- Shreyans: Did extensive experimentation on prompt engineering including zero shot, one shot and chain-of-thought prompting for both vision and extension tasks. Did in-depth source code study and performed multiple experiments with already existing libraries like AutoGPT, or Captcha solver. Provided inputs and research to the team for both of the above points which were used for the development. Made minor quality-of-life changes to the vision code (Ramachandran, 2024). Did a heavy part of report writing and documentation for the vision approach. Performed manual experimentation on the Chrome extension and wrote the readme.

- Rishab: Did a majority of the experimentation, development, and testing for the Vision approach. A significant contributor to all vision-related sections of the report. Did a little experimentation with prompt engineering.

- Jayesh: Wrote several sections of the report, experimented with zero shot, one shot, and Chain of Thought for Captcha and other web tasks, conducted tests for the Vision component as well as contributed to the vision development.

9 Conclusion

We have gained valuable insights about various techniques and challenges that come with implementing web controlling agents powered by large language models. While we believe that we are strides away from mimicking human-like behavior for solving tasks provided by the user, we believe that our work has shown significant progress and potential towards this challenge. One of the key takeaways from this project is the versatility of large language models. LLMs can seamlessly handle tasks across various domains and perform impressively out-of-the-box without any additional fine-tuning or customization. Also, effective prompt engineering is crucial when using an LLM to complete a complex task and techniques such as chain-of-thought prompting are essential in improving reliability and reducing hallucinations. Finally, combining text content with visual analysis improves an LLM's ability to understand and interact with an environment.

We knew going into the project that the goals we set for ourselves were ambitious, however, implementing a reliable Captcha solving component was more difficult than we thought. Even a simple puzzle like reading a string of text and numbers and entering it into a box was not consistent. This necessitated us to come up with creative methods such as taking advantage of the GPT-4V API and accessibility features built into reCaptcha to assist our agent in solving the challenges. Building upon this, staying covert while testing our methods on real websites proved to be an issue when we were blocked by reCaptcha for periods of time even if we tried to solve it manually.

We were surprised by how well our baseline model was able to plan out the steps it needed to take to carry out a user's request with no prompt engineering or preprocessing which again shows how advanced newer LLMs have become.

In the future we could explore features such as incorporating user feedback into our Chrome extension, allowing the agent to learn from user corrections and suggestions. This would help the agent refine its actions and outputs based on real-world interactions, making it more effective. This would also help with identifying any recurring issues and improve the agent's overall reliability. Another direction of the project we could pursue is integrating an agent flow which could determine when to use GPT-V or our DOM-based approach or even both combined depending on the environment it is in and the task it has received.

10 AI Disclosure

- Did you use any AI assistance to complete this proposal? If so, please also specify what AI you used.
  - ChatGPT 4
  - ChatGPT-4o
  - Claude

If you answered yes to the above question, please complete the following as well:

- If you used a large language model to assist you, please paste *all* of the prompts that you used below. Add a separate bullet for each prompt, and specify which part of the proposal is associated with which prompt.
  - replace the _ with words that complete the sentence: To check if our agent successfully solve Captchas, we use GPT-4V to _ for confirmation in the webpage before and after the agent executed its actions.
    * Related work
  - fix the flow and grammar of this sentence: In the future we could explore features such as incorporating user feedback into our Chrome extension where the agent learns from user corrections and suggestions.
    * Conclusion
  - is inhuman the correct word to use: The agent was flagged for inhuman activity
    * Baselines Interaction
  - can you rewrite this sentence with right english and complete my point We believe that this agent provides us with a practical tool to automate repetitive and tedious tasks with also giving us multiple technical takeaways to understand multiple things like LLM behaviour with different prompts.....
    * What you proposed vs. what youaccomplished section
  - with proposal uploaded, can you help. me start the What you proposed vs. what youaccomplished section
    * Your dataset
  - can you start the "your dataset" section, we only used the dom data.
    * Captcha solver
  - write about how we used this library, which works as follows: +———————————+ | Start run function | +———————————+ | v +———————————+ | Load Agents | +———————————+ | v +———————————+ | Generate prompt for image check | +———————————+ | v +———————————+ | Check if image looks like captcha | +———————————+ | +———————————+ No | | Yes v | +———————————-+ v | Raise ValueError | +———————————+ | | Generate prompt for captcha type | | +———————————+ | | | v | +———————————+ | | Determine captcha type | | +———————————+ | | | +———-+———+ | | | | | v v | | +———+–+ +—+———+ | | | Text captcha| | Math captcha| | | +———+–+ +—+———+ | | | | | | v v | | +———+–+ +—+———+ | | | Solve text | | Solve math | | | | captcha | | captcha | | | +———+–+ +—+———+ | | | | | | v v | | +———+–+ +—+———+ | | | Return text | | Return math | | | | captcha | | captcha | | | | solution | | solution | | | +———+–+ +—+———+ | | | | | | +———-+———+ | | | +———————————+ | +———————————v———————————+ | Raise ValueError if captcha type is unsupported | +———————————+ | v +———————————+ | End run function | +———————————+
  - how do I add page numbers to this - the start of the paper latex code
  - **Free response:** For each section or paragraph for which you used assistance, describe your overall experience with the AI. How helpful was it? Did it just directly give you a good output, or did you have to edit it? Was its output ever obviously wrong or irrelevant? Did you use it to generate new text, check your own ideas, or rewrite text?
    * Related work - It was helpful and directly gave me a good answer but I edited it to make it more concise. The output was not wrong and I used it to generate new text.
    * Conclusion - It was helpful and directly gave me a good answer but I edited it. The output was not wrong and I used it to rewrite text.
    * Baselines Interaction - It was helpful and directly gave me a good answer. The output was not wrong and I used it to rewrite text.
    * Approach - It was helpful in getting a good skeleton to follow and used it to get a draft
    * Misc : Tried to add pagenumbers but didnt really get a good response, and the answer did not work.
    * Captcha solver : We used the flow

from the github to get a draft written for the captcha solver github part.

References

Bock, K., Patel, D., Hughey, G., and Levin, D. (2017). unCaptcha: A Low-Resource defeat of reCaptcha's audio challenge. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC. USENIX Association.

Crosby, P. and Sukhar, I. (2009). Vimium: The hacker's browser.

Deng, G., Ou, H., Liu, Y., Zhang, J., Zhang, T., and Liu, Y. (2024). Oedipus: Llm-enchanced reasoning captcha solver.

Mernik, M., Heering, J., and Sloane, A. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–.

Muthusamy, V., Rizk, Y., Kate, K., Venkateswaran, P., Isahagian, V., Gulati, A., and Dube, P. (2023). Towards large language model-based personal agents in the enterprise: Current trends and open problems. pages 6909–6921.

Nass, M., Alegroth, E., and Feldt, R. (2023). Improving web element localization by using a large language model.

OpenAI (2024). Openai api documentation. Accessed: 2024-05-15.

Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., and Sutskever, I. (2023). Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning*, pages 28492–28518. PMLR.

Ramachandran, A. (2024). Aashiqramachandran/i-am-a-bot: An multi-modal llm powered agent to automatically solve captchas.

Sridhar, A., Lo, R., Xu, F. F., Zhu, H., and Zhou, S. (2023). Hierarchical prompting assists large language model on web navigation.

thicccat688 (2023). Recaptcha solver for selenium (using audio).

Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. (2024). Executable code actions elicit better llm agents.

Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., Zheng, R., Fan, X., Wang, X., Xiong, L., Zhou, Y., Wang, W., Jiang, C., Zou, Y., Liu, X., Yin, Z., Dou, S., Weng, R., Cheng, W., Zhang, Q., Qin, W., Zheng, Y., Qiu, X., Huang, X., and Gui, T. (2023). The rise and potential of large language model based agents: A survey.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2023). React: Synergizing reasoning and acting in language models.

Zhang, X., Lu, Y., Wang, W., Yan, A., Yan, J., Qin, L., Wang, H., Yan, X., Wang, W. Y., and Petzold, L. R. (2023). Gpt-4v(ision) as a generalist evaluator for vision-language tasks.

Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B. Y., Fu, J., Chen, W., and Yue, X. (2024). Opencodeinterpreter: Integrating code generation with execution and refinement.