WebWISE: Unlocking Web Interface Control for LLMs via Sequential Exploration

Heyi Tao[1] * **Sethuraman T V**[1]* **Michal Shlapentokh-Rothman**[1]
Tanmay Gupta [2] **Heng Ji**[1] **Derek Hoiem**[1]
[1]University of Illinois at Urbana Champaign [2]PRIOR @Allen Institute for AI
{heyitao2,st34,michal5,hengji,dhoiem}@illinois.edu
tanmayg@allenai.org

Abstract

This paper investigates using Large Language Models (LLMs) to automatically perform web software tasks using click, scroll, and text input operations. Previous approaches, such as reinforcement learning (RL) or imitation learning, are inefficient to train and task-specific. Our method uses filtered Document Object Model (DOM) elements as observations and performs tasks step-by-step, sequentially generating small programs based on the current observations. We use in-context learning, either benefiting from a single manually provided example, or an automatically generated example based on a successful zero-shot trial. We evaluate our proposed method on the MiniWob++ benchmark. With only one in-context example, our WebWISE method using gpt-3.5-turbo achieves similar or better performance than other methods that require many demonstrations or trials.

1 Introduction

A major goal of AI is to develop intelligent agents that interact with their environments to perform tasks. This goal is often explored in the context of physical environments. Our work explores performing software tasks with the long-term aim of creating agents that work using software designed for humans. Software tasks are valuable by themselves — much of the work we do is on computers — and also offer highly controllable and repeatable tasks that have many of the same challenges as physical tasks, such as manipulable environments, goals that require long sequences of actions, and need for exploration. Prior works to control software have used reinforcement learning (RL), requiring many demonstrations and scored trials to learn simple interaction tasks. Instead, we use Large Language Models (LLMs) to generate actions (e.g., click and enter text) based on environment observations (DOM elements) in web software.

The goal is to complete a web software task, given a natural language instruction and an API for observing and interacting with the environment. In this setting, our observations are DOM elements, indicating the layout and state of buttons, text, and other displayed elements. Actions include clicking an element, scrolling the mouse wheel, and entering text in a text box. The experiments are performed on the MiniWoB++ benchmark (Liu et al., 2018), which consists of randomized simple tasks that involve menu navigation, text entry, and/or clicking buttons and other interactive elements.

LLMs are commonly used to generate code, e.g., Codex (Chen et al., 2021), and their use has recently been explored to complete AI tasks given an API and a small number of in-context examples, e.g., VisProg (Gupta and Kembhavi, 2023), ViperGPT (Surís et al., 2023), ProgPrompt (Singh et al., 2023), Code4Struct (Wang et al., 2023), and Vi-Struct (Chen et al., 2023). Our application of controlling software differs in two key ways:

- **Environment Grounding**: Interacting with software requires knowledge of the environment, such as the layout of elements on the screen. For automatic interaction with web software, we propose to extract and filter pertinent information from the DOM elements as the environmental observations.
- **Sequential Decision-Making**: As the environment responds dynamically to actions, we take a sequential approach to generate actions rather than creating an entire actions sequence all at once. This allows us to generate actions informed by current observations while maintaining previous action-observation pairs.

An additional challenge is how to train LLMs to better control software. This study limits the focus

*Equal Contribution

Findings of the Association for Computational Linguistics: NAACL 2024, pages 3693–3711

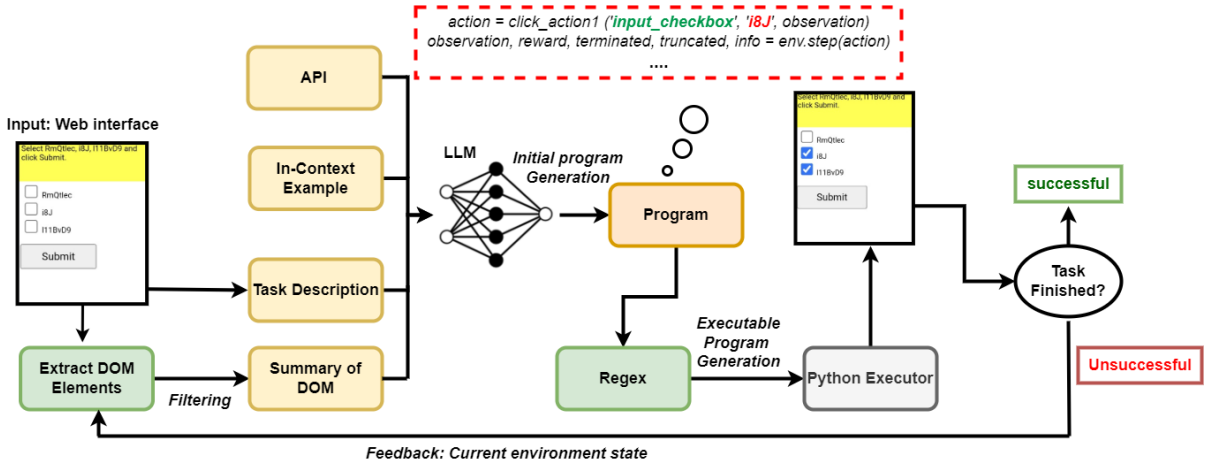

Figure 1: We show the path of an agent in the WebWISE method. The representation of the components in the diagram is as follows: yellow boxes represent inputs to the LLM, the gray box represents executor, Unshaded red/green boxes symbolize final rewards/output, and shaded green boxes denote Python functions.

to in-context learning. One option is to manually create examples of programs that satisfy instructions, but this requires some level of expertise and experimentation to be effective. Observing that the LLMs occasionally produce successful programs for specific tasks sometimes (rather than always or never), we propose using these successes as a form of context. This way, LLMs can create its own context using only a success indicator.

In this paper, our **main contribution** is to investigate the use of LLMs to control web software, particularly:

- Use of filtered DOM elements as web software observations, which we find outperforms more comprehensive read-outs.
- Effects of an iterative approach of cycling action and observation, which we show outperforms one-shot generation of an action sequence.
- Auto-generating context based on successful zero-shot trials, that outperforms zero-shot performance in many cases and requires no programming or knowledge of the control API.

2 Related Work

2.1 Automated Computer Tasks

Most methods for automating computer tasks use some form of RL. Common approaches, including Q-Learning (Jia et al., 2019), imitation learning (Yao et al., 2022), and policy learning and behavioral cloning (Zheng et al., 2021) have achieved human-level performance on the web interface benchmark MiniWob++ (Liu et al., 2018). Other approaches, such as (Humphreys et al., 2022; Zhong et al., 2022), combine RL with other modalities. CCNet (Humphreys et al., 2022) is a multimodal architecture specifically designed for automating software and is trained using a combination of RL and imitation learning. Language Dynamics Distillation (Zhong et al., 2022) is pretrained by predicting environment information and fine-tuned with RL. WebGUM (Furuta et al., 2023) is trained by combining finetuning for an instruction-finetuned model with a vision encoder on a large number of demonstrations. While these methods work well, they often require thousands of demonstrations and/or millions of trials. We investigate the use of LLMs with only a single in-context example per type of task, towards creating an approach that can be easily extended and adopted.

2.2 Reasoning and Action in Large Language Models

Recent efforts explore applying LLMs to decision-making (Mialon et al., 2023) and reasoning (Huang and Chang, 2023). Initial work focused on how to convert natural language output to admissible actions (Huang et al., 2022a). Huang et al. use a Bert LM (Devlin et al., 2019) model pre-trained with SentenceBert (Reimers and Gurevych, 2019) to directly convert the output of GPT-3 (Brown et al., 2020) to an executable action. SayCan (Ahn et al., 2022) uses an alternative approach where for each action, the probability of generating that action using a language model is multiplied by the action's value function. Inner Monologue (Huang

et al., 2022b) builds on SayCan (Ahn et al., 2022) by introducing feedback from the environment. Unlike these approaches, our web environment is much simpler, and we found that with specific prompting, we can convert natural language output to an appropriate action. Prompting, in particular chain-of-thought prompting, has been used to demonstrate LLM's reasoning ability (Wei et al., 2022; Kojima et al., 2022; Nye et al., 2021). A related area of research combines combining reasoning and decision-making skills into one method. SayCan (Ahn et al., 2022) and Inner Monologue (Huang et al., 2022b) are both early examples of such a combination. ReAct (Yao et al., 2023) expands these works by adding language (generated by an LLM) to the list of possible actions. After each action is executed, a thought (language action) is generated based on the previous action and environment. Reflexion (Shinn et al., 2023) builds upon ReAct (Yao et al., 2023) by allowing access to previous actions and states.

2.3 Visual Programming

In this work, we leverage LLMs' program generation ability (Chen et al., 2021) to control web-based software. VISPROG (Gupta and Kembhavi, 2023) introduces the idea of visual programming, where programs call APIs to interpret and transform images using pre-trained models, solving tasks like image editing and Visual Question Answering (VQA). VISPROG was one of the first approaches demonstrating the capability of LLMs to solve a wide array of vision tasks effectively, serving as an inspiration for our work. VISPROG and related works (Surís et al., 2023; Wu et al., 2023; Gao et al., 2023; Wang et al., 2022) use prompts containing APIs, example programs, and instructions to guide LLMs in tasks like image editing and Visual Question Answering (VQA).

Visual programming methods produce impressive zero-shot results but are limited in that they generate one-shot programs without observing the image/environment. Our approach generates programs in multiple steps and uses DOM elements to summarize the visual input. Methods like HeaP (Sodhi et al., 2023) learn a set of hierarchical LLM prompts for planning high-level tasks and executing low-level policies. While HeaP is an innovative approach, it uses training data collected from users. Such a reliance might pose a challenge in scenarios where demonstrations are scarce or the tasks are highly dynamic and personalized. The approach 'Recursively Criticizes and Improves (RCI)' (Kim et al., 2023), a concurrent study, similarly employs LLMs for software interaction. RCI takes HTML as observations and incorporates up to 22 in-context examples. Moreover, their examples are specific to the variations of each task. In contrast, we focus on assessing the efficiency of in-context examples, providing only one per type of task.

3 MiniWob++ Benchmark

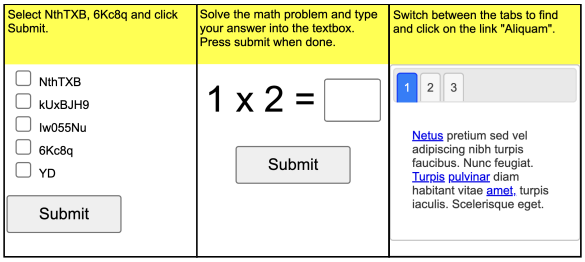

Figure 2: Screenshots of example tasks from MiniWob++ benchmark. Each task contains a natural language instruction at the top and a task interface to perform primitive actions in the bottom.

Our objective is to generate programs to control web interaction tasks. To evaluate the effectiveness of these generated programs, we use the MiniWob++ (Liu et al., 2018) benchmark that captures the salient challenges of browser interactions in a simple setting. This web-based simulation environment is an extension of MiniWob (Shi et al., 2017) originally introduced by OpenAI. The MiniWob++ benchmark contains more than 100 tasks of varying difficulty levels, and we chose 48 of those tasks to evaluate our methods on. Each MiniWob++ task contains a natural language instruction and an HTML/DOM representation of the web page containing the task. An agent can interact with MiniWob++ through the Selenium API. Successfully completed tasks receive a score of "1"; otherwise, they receive a score of "-1". We describe the actions we use in the section 4. A few of the tasks we selected for this study are shown in Figure 2.

4 Methods

This section describes our problem formulation and progressive layers to our approach: based on instruction alone, incorporating filtered DOM elements as observations, acting step-by-step, and auto-generating context. Figure 1 and Algorithm 1 show an overview of our full approach, which we call WebWISE.

Algorithm 1 WebWISE Function

procedure WEBWISE(**input** Task Description $\mathcal{L}$, API a, in-context example $\mathbf{e} = e_1, \ldots, e_m$, program generator π, DOM elements D_t)
  Initialize success state s as False, iteration count t as 0, maximum iterations as T
  while $s = False$ **and** $t < T$ **do**
    $d_t = getSummary(D_t)$ ▷ Extract relevant DOM elements
    $x_t = \pi(\mathcal{L}, d_t, a, \mathbf{e})$ ▷ Generate program
    $s = \phi(x_t, d_t)$ ▷ Execute program
    $D_{t+1} = getDOM()$ ▷ Update DOM elements after execution
    $t = t + 1$
  end while
end procedure

4.1 Problem Formulation

Each task involves a virtual agent assigned to carry out high-level natural language instructions (task descriptions) denoted as $\mathcal{L}$. The agent must interact with the environment, which is initially represented by a set of observable DOM Elements D_t. A program generator π is employed to generate program $x_t = \pi(\mathcal{L}, d_t)$. The execution engine ϕ then applies the generated program through $\phi(x_t, d_t)$, resulting in success indicator s and updated DOM Elements D_{t+1}, representing the environment's next state. A maximum number of iterations T is set to limit the total number of iterations that the LLM can generate programs.

4.2 DOM Elements

The choice among DOM elements, HTML, and RGB values when interacting with a web interface largely hinges on the task's demands, the intricacy of the webpage, and the agent's capabilities. In this work, DOM elements were used due to their simplicity and structured nature. Models such as Pix2Struct(Lee et al., 2023) convert webpage screenshots into structured HTML when direct DOM access is not possible. The DOM elements allow the agent to engage directly with the page – clicking on elements, inputting text into form fields, reading text from the page, and so forth – and is generally simple to implement and comprehend. The full text of the DOM elements can contain items not crucial for a particular task, so we use a simple filtering function $getSummary$ that returns a subset of the current DOM elements that belong to a pre-defined list of "tags" and "classes". Further details are in the appendix.

4.3 Single-Step Approach

In the single-step approach, one program is generated and executed for a given task. We use gpt-3.5-turbo (Ouyang et al., 2022) and Llama-2 7B (Touvron et al., 2023) as our program generator π. The input to the LLM is a prompt with 4 parts: filtered DOM elements, API a, task description, and an in-context example e. Our API a has three basic functions: *click*, *enter text*, and *scroll*. Each executes actions within MiniWob++ (Liu et al., 2018). A summary of our API can be seen in Listing 1 and is constant across all tasks. A complete listing of our APIs is shown in the appendix Listing 7. Hand-crafted in-context examples were used like in GPT-3, which can be seen in Figure 17 in the appendix. A sample task input is given followed by the expected output program.

```
def getSummary(dom_elements):
    '''
    Input: DOM elements
    Output: Subset of DOM elements
    '''
def click_action1(tag_class_name,id_text_name,observation):
    '''
    Input: tag or element, id or text, observation
    Output: clicks on a specific element in the environment
    '''
def enter_text_action(input_text,observation):
    '''
    Input: text, observation
    Output: enters text in element in the environment
    '''
def scroll_action1(text_to_scroll_to,observation):
    '''
    Input: text, observation
    Output: moves webpage such that certain text is visible
    '''
```

Listing 1: Summary of our API

The generated program x_t is a string of Python code, executed using the execution engine ϕ. However, the code generated might not be executable or use the API a correctly. A regular expression is used to extract the executable code and filter the irrelevant parts of the generated output. If the generated code proves entirely unusable, this particular program is skipped, and the LLM will move to the next iteration of generating programs to solve this task. If usable code is identified, $\phi(x_t, d_t)$ will be executed, and the LLM will rely on the environmental feedback to determine whether the task has been successfully completed. The iterations con-

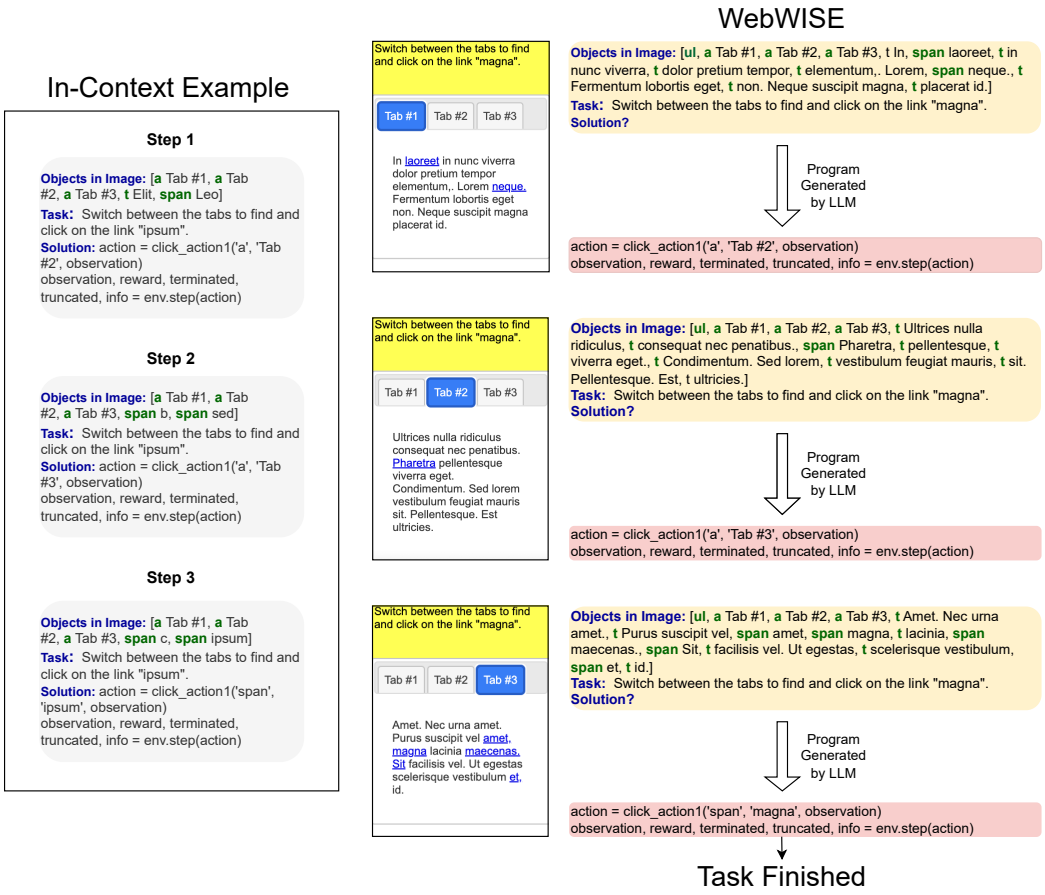

Figure 3: An example of WebWISE for the 'click-tab' task. The objective is to click on link that is not present within the initial DOM elements. At each step, the environment updates, and a new program is generated based on the context. The left part of the image illustrates the step-by-step in-context example with fictional DOM elements and tasks. This example guides LLMs in generating a program for the current task interface, executing one action at a time. The middle and right parts of the image shows the task and solution generated by LLMs.

tinue until either the task is successfully completed (i.e., $s = \text{True}$), or the number of iterations reaches the predefined maximum ($t = T$).

4.4 Multi-Step Approach

Our multi-step method, referred to as WebWISE, is illustrated in Algorithm 1. In contrast to single-step generation, WebWISE generates and executes programs incrementally until the environment signals that the task has been solved correctly or incorrectly. If the task has not been solved after a particular number of iterations (T), the environment signals a failure (by returning -1). The DOM elements from the ith iteration becomes the initial DOM elements for the $(i+1)$th iteration. This approach is employed in both zero-shot and one-shot scenarios. In the zero-shot scenario, an additional instruction (task message) is provided to the LLMs, prompting it to generate programs progressively, based on changes in the DOM elements throughout the task. In the one-shot scenario, the in-context examples shift at each step, coupled with the additional instruction (task message) to 'perform in a step-by-step fashion', akin to the zero-shot scenario. Ablation studies concerning the sensitivity of the task messages are included in the appendix. Figure 3 illustrates an example of the WebWISE methodology. Conditioning the program generation on the current environment simplifies the execution of sequential tasks. This is because the model can observe the effect of its actions in the environment through the changes of DOM elements, and consequently generates more effective actions for the next step. In contrast, the single-step approach generates the entire program based on the initial set of DOM elements.

4.5 Automatic Generation of In-Context Examples From Scored Trials

Our empirical findings reveal improved success rates when an in-context example is provided. Each in-context example includes an observation, task description, and a program that would satisfactorily complete the task according to the task description. However, supplying such examples requires a comprehensive understanding of the API and programming proficiency, which can pose an obstacle for intricate tasks or novice users. Prompted by these challenges, we design a simple approach called Auto-Context Generation (ACG) to automatically generate an in-context example from a successful trial. To develop ACG, we conduct a series of zero-shot trials. In each trial, a program is generated and executed in the absence of any in-context examples. If the program executes the task correctly, it is stored, along with the original task description and filtered DOM elements, as the in-context example. After 10 zero-shot trials, the single-step approach (4.3) is applied for 50 iterations. The correctly generated programs during the trial stage serve as in-context examples, and only two such programs are preserved at maximum. When appending the in-context examples to the prompt, a specific statement:"*Here is one example you have solved with a successful solution.*" is also included. While our experiments use this fully automated approach, a user, in a practical context, may be able to guide the successful completion using feedback and prompts to acquire the in-context example.

5 Experiments and Results

All approaches (single-step, multi-step and Auto-Context Generation) are evaluated on 48 tasks derived from the MiniWob++ dataset (Liu et al., 2018). These tasks are chosen to span a range of complexities. Additionally, a variant of the single-step method, which excludes any DOM elements, is evaluated and labeled as the 'Instruction Only' method. The conventional single-step process is labeled as 'Instruction+ Filtered DOM' approach. We evaluate our methods using gpt-3.5-turbo and Llama 2 7B. However, we primarily focus on gpt-3.5-turbo as the model demonstrates significantly better performance.

5.1 Implementation details

For all experiments, the temperature was set to 0 and the input token limit was 4096 for the gpt-3.5-turbo with training data up to September 2021. We used the Llama-2 7B implementation from HuggingFace with 8-bit quantization on a single RTX 3090 Ti. The same prompt is applied at the beginning of each method, which can be found in the appendix material. In addition to evaluating each of the 48 tasks with a single in-context example ($k = 1$), we also evaluate the zero-shot setting ($k = 0$). Tasks are scored as follows: "1" for successful completion and "-1" for failure. For the 'Instruction-Only', 'Instruction+DOM', and 'WebWISE' (multi-step) approaches, we executed each task for 50 iterations and averaged the success over the 50 iterations. For the 'Auto-Context Generation' method, we initially carried out 10 zero-shot trials, followed by 50 iterations of the 'Instruction+DOM' (single-step) method.

5.2 Results

We categorized various tasks from the MiniWob++ dataset (Liu et al., 2018) based on the number of predefined function calls necessary to accomplish the task. The groups include tasks requiring 1 function, 2 functions, between 3 and 6 functions, and a variable number of functions. The detailed task classification is available in Appendix D.1.

Table 2 summarizes the average success rate of our proposed methods, and Table 3 compares our methods with other prior reinforcement learning (RL) and behavior cloning (BC) based approaches. Results from prior methods were grouped into the same categories as our tasks. For simple tasks which requires just one function call, our approach with gpt-3.5-turbo outperforms WebNT5-3B (Gur et al., 2022) benchmark by a slight margin of 14.1% which employs a finetuned large language model with 12K expert demonstration data. WebWise using gpt-3.5-turbo outperforms WebN-T5-3B, CCNet (RL) (Humphreys et al., 2022), and CCNet (BC) (Humphreys et al., 2022). CCNet (BC+RL) (Humphreys et al., 2022) significantly outperforms our approach, but requires many expert demonstrations and millions of RL trials, while our approach requires minimal per-task learning. Table 1 shows the success rate for all evaluated tasks across different methods including RCI and HeaP. While the results suggest that RCI and HeaP exhibit marginally superior performance in certain tasks, this difference could be attributed to the number of in-context examples in RCI since up to 22 examples were used. Additionally, HeaP adopts a chain-of-thought (Wei et al., 2022) prompting,

which typically surpasses standard prompting techniques. HeaP also uses a currently depreciated model instruction-tuned text-davinci-003 (OpenAI, 2024) instead of gpt-3.5-turbo, which may also cause minor performance differences.

5.3 Ablation

Table 2 compares our proposed methods across the task groups for both zero-shot (k=0) and single-shot (k=1). For a particular value of k, 'Instruction Only' has the lowest performance though the gap between 'Instruction Only' and the other methods grows as the task become more complex (left to right). For the easiest tasks, using DOM elements and feedback (WebWISE) has little effect on the performance. For k=1, WebWISE produces the largest gain in the Variable Function group. Auto-Context Generation has a similar or higher performance than the other zero-shot methods but is lower than the single-shot ones.

6 Discussion

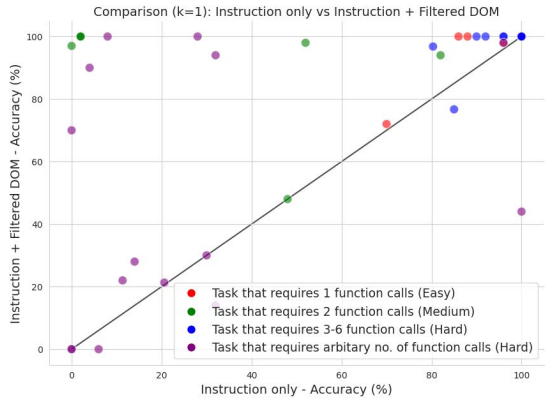

Figure 4: Comparison of k=1 performance across all tasks for Instruction Only (x-axis) and Instruction+Filtered DOM (y-axis).

Drawing from prior work that demonstrates the ability of LLMs to perform computer vision and embodied tasks, our work aims to extend the application of LLMs to more intricate challenges, specifically web interface tasks. In doing so, we shifted the focus from reliance on multiple learning examples to an approach based on zero and one-shot learning. Key insights are presented from Table 2, Figure 4, 5, 6, and 7 based on gpt-3.5-turbo results.

Influence of Single In-Context Example and DOM Elements: We explored the impact of a single in-context example, represented as k=1, across several task groups. The results indicate the ability

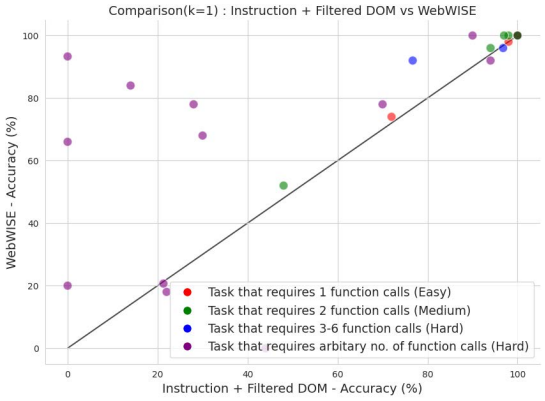

Figure 5: Comparison of k=1 performance across all tasks for Instruction+Filtered DOM (x-axis) and WebWISE (y-axis).

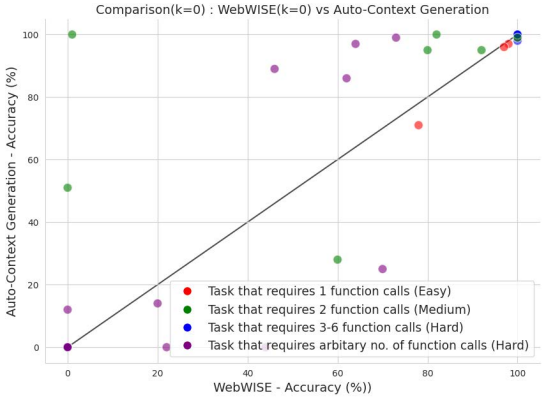

Figure 6: Comparison of k=0 performance across all tasks for WebWISE(x-axis) and Auto-Context Generation (y-axis).

of a LLM to perform well based on a single correct example. As shown in Figure 4, most easy tasks can be successfully completed even without observations (DOM elements). However, as the task complexity increases, the inclusion of observations becomes critical which is evident from the results.

Improvement from Step-by-Step Actions: Both Table 2 and Figure 5 clearly illustrate that implementing actions and observations in a sequential manner (WebWISE) significantly enhances performance compared to the 'single-step' action generation. This improvement is especially noticeable for more challenging tasks, where many tasks require the execution of actions in a specific sequence and changes in the environment. When k=1, step-by-step outperforms single-step by a large margin (23% to 75% success rate).

Performance of Auto-Context Generation (ACG): Figure 6 shows that auto-context examples increase success rates, compared to no in-context

Task	Accuracy										
	Instruction Only		Instruction+ Filtered DOM		Instruction+ Whole DOM		WebWISE		Auto-Context Generation	Related Works	
	k=0	k=1	k=0	k=1	k=0	k=1	k=0	k=1	Zero Shot Trials=10	RCI	HeaP
click-button-sequence	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
click-button	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
click-checkboxes-large	0.00	0.04	0.26	0.90	TLE	TLE	0.46	1.00	0.89	0.94	1.00
click-checkboxes-soft	0.00	0.00	0.61	0.70	0.18	0.41	0.62	0.78	0.86	0.72	0.54
click-checkboxes-transfer	0.04	0.28	0.66	1.00	0.52	TLE	0.73	1.00	0.99	1.00	0.94
click-checkbox	0.08	0.08	0.58	1.00	0.51	1.00	0.64	1.00	0.97	1.00	0.90
click-collapsible-2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.62	0.80
click-collapsible	0.00	.002	1.00	1.00	0.96	1.00	1.00	1.00	1.00	1.00	1.00
click-dialog-2	0.64	0.70	0.70	0.72	0.68	0.54	0.78	0.74	0.71	1.00	1.00
click-dialog	0.00	1.00	1.00	1.00	0.88	1.00	1.00	1.00	1.00	1.00	1.00
click-link	0.00	0.88	0.96	1.00	0.88	0.92	0.98	1.00	0.97	1.00	1.00
click-option	0.18	0.02	0.74	1.00	0.70	TLE	0.82	1.00	1.00	1.00	1.00
click-pie	0.00	0.00	0.00	0.00	0.00	0.00	0.74	0.80	0.52	-	1.00
click-tab-2-hard	0.00	0.30	0.14	0.30	TLE	TLE	0.22	0.68	0.00	0.76	1.00
click-tab-2-easy	0.00	0.82	0.92	0.94	0.84	0.86	0.92	0.96	0.95	-	-
click-tab-2-medium	0.00	0.48	0.36	0.48	TLE	TLE	0.60	0.52	0.28	-	-
click-tab-2	0.00	0.14	0.04	0.28	TLE	TLE	0.44	0.78	0.00	0.74	1.00
click-tab	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	-
click-test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
click-test-transfer	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	-	-	
click-test-2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
click-widget	0.02	0.96	0.96	0.98	0.94	1.00	0.97	0.98	0.96	0.98	1.00
enter-date	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.96	1.00
enter-password	0.00	0.96	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
enter-text-dynamic	0.00	0.96	1.00	1.00	1.00	1.00	1.00	1.00	0.98	1.00	1.00
enter-text-2	0.00	0.90	1.00	1.00	0.96	0.96	1.00	1.00	1.00	-	1.00
enter-text	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
enter-time	0.00	0.52	0.00	0.98	0.00	0.80	0.00	1.00	0.51	1.00	-
focus-text-2	0.00	1.00	1.00	1.00	0.78	1.00	1.00	1.00	1.00	1.00	1.00
focus-text	0.00	0.92	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	-
guess-number	0.00	0.32	0.08	0.14	0.00	0.00	0.20	0.84	0.14	0.20	-
login-user	0.00	0.96	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
grid-coordinate	0.00	0.86	0.00	1.00	0.00	TLE	0.00	1.00	0.00	1.00	1.00
multilayout	0.00	0.00	0.00	0.64	0.00	0.00	0.00	0.78	0.00	0.72	0.94
read-table	0.00	0.00	0.28	0.48	0.24	0.30	0.80	0.86	0.90	-	-
read-table-2	0.00	0.00	0.20	0.32	0.16	0.22	0.80	0.82	0.88	-	-
simple-arithmetic	0.00	0.00	0.90	0.97	0.90	0.90	1.00	1.00	0.99	-	1.00
simple-algebra	0.00	0.02	0.55	1.00	0.84	0.90	0.80	1.00	0.95	1.00	0.74
navigate-tree	0.00	0.32	0.51	0.94	0.60	0.74	0.70	0.92	0.25	0.86	-
search-engine	0.00	0.06	0.00	0.00	0.00	0.00	0.00	0.20	0.00	-	1.00
find-word	0.00	1.00	0.00	0.44	0.00	0.18	0.00	0.00	0.00	-	0.98
email-inbox-forward-nl-turk	0.00	0.85	0.00	0.77	0.00	0.00	0.00	0.92	0.00	0.94	0.90
email-inbox-forward-nl	0.00	0.80	0.00	0.97	0.00	0.00	0.00	0.96	0.00	1.00	0.74
email-inbox-nl-turk	0.00	0.11	0.00	0.22	0.00	0.00	0.00	0.18	0.00	0.98	1.00
email-inbox	0.00	0.20	0.00	0.21	0.00	0.00	0.00	0.21	0.00	0.98	0.90
terminal	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	-
text-transform	0.00	0.00	0.00	1.00	1.00	1.00	0.00	1.00	0.00	-	-
use-autocomplete	0.00	0.80	0.84	0.92	0.80	0.92	0.92	0.92	0.90	-	-

Table 1: Comparison of performance across tasks of our methods and RCI using the gpt-3.5-turbo and HeaP using instruction tuned text-davinci-003. TLE (Token Limit Exceeded) signifies tasks where the full prompt exceeds the LLM's context window. "-" denotes tasks not addressed by RCI and HeaP methods.

examples, for the large majority of cases, including all levels of difficulty. However,as Table 2 shows, a single manually provided in-context example leads to even higher success rates, especially for the hardest tasks. While ACG is comparable to the other zero-shot methods, it still lags behind the single-shot methods across different tasks. This indicates that either the in-context examples generated are not as effective as hand-crafted ones or no in-context examples were generated at all.

Impact of filtered DOM elements: Figure 7 shows that filtering the DOM elements improves success rates for the large majority of tasks, as the LLM is able to focus on more relevant information. However, Figure 7 highlights several tasks where the performance of our approach when provided

Methods	1 Function		2 Function		3-6 Function		Variable Function	
	k=0	k=1	k=0	k=1	k=0	k=1	k=0	k=1
Instruction Only	0.45	0.94	0.03	0.27	0.00	0.94	0.01	0.20
Instruction + Whole DOM	0.82	0.94	0.71	0.89	0.77	0.80	0.16	0.23
Instruction + Filtered DOM	0.86	0.97	0.64	0.91	0.80	0.97	0.21	0.44
WebWISE	0.87	0.97	0.73	0.93	0.80	0.99	0.27	0.75
Auto-Context Generation	0.86	-	0.81	-	0.80	-	0.30	-

Table 2: Average success rate across tasks for different versions of our approach evaluating on gpt-3.5-turbo, with WebWISE and Auto-Context Generation (ACG) being the main approach. k=0 means no in-context example is provided, or in case of ACG, only auto-context examples are provided; k=1 indicates one manual in-context example is provided, though many examples are shared across tasks.

Methods	1 Function		2 Function		3-6 Function		Variable Function	
	k=0	k=1	k=0	k=1	k=0	k=1	k=0	k=1
WebWISE(gpt-3.5-turbo)	0.87	0.97	0.73	0.93	0.80	0.99	0.27	0.75
WebWISE(Llama 2 7B)	0.45	0.76	0.03	0.24	0.28	0.41	0.03	0.18
WebNT5-3B	0.83		0.29		0.73		0.37	
WebN-T5-3B(k=0)	0.85		0.27		0.63		0.30	
CCNet(BC+RL)	0.99		0.94		0.99		0.89	
CCNet(RL)	0.88		0.65		0.50		0.44	
CCNet(BC)	0.77		0.37		0.27		0.16	
WebGUM(HTML)	0.92		0.40		1.00		0.83	
WebGUM(HTML+Image)	0.94		0.40		1.00		0.90	

Table 3: Comparison of our WebWISE method on different LLMs with methods reported from other works where the value of k is not applicable. CCNet has been trained using behavior cloning (BC) on human-labeled data and also reinforcement learning by interacting with MiniWob++ different tasks environment. CCNet (BC) represents the model that has been only trained on human-labeled data, CCNet (RL) is trained only by letting it interacting with MiniWob++ environment for many trials, and CCNet (BC+RL) is trained using both methods.

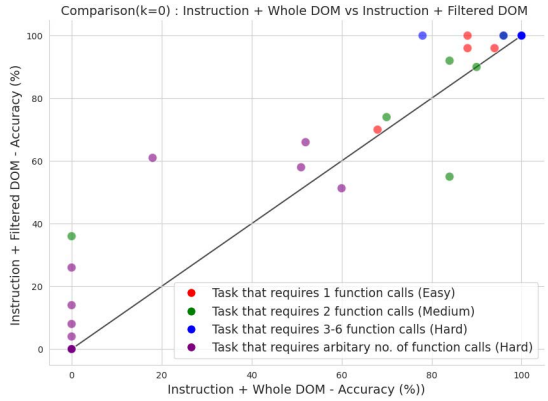

Figure 7: Comparison of k=0 performance across all tasks for Instruction+Whole DOM(x-axis) and Instruction+Filtered DOM(y-axis).

with whole DOM elements is better than the filtered DOM elements. This is likely due to some information important contained in the Whole DOM elements. Thus, developing an adaptive DOM element filter that can generalize across several tasks could be a next step. The influence of different LLMs on our results are discussed in appendix.

Limitations: Our main limitations include that experiments are limited to simple tasks, sensitivity to the input prompts, lack of an explicit memory, and use of only two LLMs with only one being open-source. Eventually, we aim to develop systems that can learn to perform more complicated tasks, like booking airline tickets, with few trials or demonstrations. This requires being able to more fully utilize web interfaces and retain memory of past interactions to complete long action sequences. Increased robustness to input prompts is also needed. Further improvement may be possible by learning from failures and automatically correcting mistakes.

7 Conclusion

Our work presents an initial exploration into using Large Language Models (LLMs) to generate programs that interact with web interfaces. Our experiments indicate: filtered DOM elements are effective forms of observation; the step-by-step action and observation is more effective than single-step generation; and automatically generated in-context examples from successful trials can boost success rates for many tasks.

Acknowledgement This work is supported in part by ONR awards N00014-21-1-2705 and N00014-23-1-2383.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, et al. 2022. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Yangyi Chen, Xingyao Wang, Manling Li, Derek Hoiem, and Heng Ji. 2023. Vistruct: Visual structural knowledge extraction via curriculum guided code-vision representation. In *Proc. The 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP2023)*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Hiroki Furuta, Ofir Nachum, Kuang-Huei Lee, Yutaka Matsuo, Shixiang Shane Gu, and Izzeddin Gur. 2023. Multimodal web navigation with instruction-finetuned foundation models. *arXiv preprint arXiv:2305.11854*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14953–14962.

Izzeddin Gur, Ofir Nachum, Yingjie Miao, Mustafa Safdari, Austin Huang, Aakanksha Chowdhery, Sharan Narang, Noah Fiedel, and Aleksandra Faust. 2022. Understanding html with large language models. *arXiv preprint arXiv:2210.03945*.

Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards reasoning in large language models: A survey. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 1049–1065, Toronto, Canada. Association for Computational Linguistics.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022a. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. 2022b. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*.

Peter C Humphreys, David Raposo, Tobias Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Adam Santoro, and Timothy Lillicrap. 2022. A data-driven approach for learning to control computers. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 9466–9482. PMLR.

Sheng Jia, Jamie Ryan Kiros, and Jimmy Ba. 2019. DOM-q-NET: Grounded RL on structured language. In *International Conference on Learning Representations*.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2023. Language models can solve computer tasks. *arXiv preprint arXiv:2303.17491*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*.

Kenton Lee, Mandar Joshi, Iulia Turc, Hexiang Hu, Fangyu Liu, Julian Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. 2023. Pix2struct: Screenshot parsing as pretraining for visual language understanding.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, and Percy Liang. 2018. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations*.

Grégoire Mialon, Roberto Dessi, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Roziere, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented language models: a survey. *Transactions on Machine Learning Research*. Survey Certification.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*.

OpenAI. 2024. Openai api models. `https://platform.openai.com/docs/models`. Accessed: 2024-03-31.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *CoRR*, abs/1908.10084.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. 2017. World of bits: An open-domain platform for web-based agents. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3135–3144. PMLR.

Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2023. Progprompt: Generating situated robot task plans using large language models. In *nternational Conference on Robotics and Automation (ICRA)*.

Paloma Sodhi, S. R. K. Branavan, and Ryan McDonald. 2023. Heap: Hierarchical policies for web actions using llms.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Xingyao Wang, Sha Li, and Heng Ji. 2022. Code4struct: Code generation for few-shot structured prediction from natural language. *arXiv preprint arXiv:2210.12810*.

Xingyao Wang, Sha Li, and Heng Ji. 2023. Code4struct: Code generation for few-shot event structure prediction. In *Proc. The 61st Annual Meeting of the Association for Computational Linguistics (ACL2023)*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. 2023. Visual chatgpt: Talking, drawing and editing with visual foundation models. *arXiv preprint arXiv:2303.04671*.

Shunyu Yao, Howard Chen, John Yang, and Karthik R Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic web testing using curiosity-driven reinforcement learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 423–435. IEEE.

Victor Zhong, Jesse Mu, Luke Zettlemoyer, Edward Grefenstette, and Tim Rocktäschel. 2022. Improving policy learning via language dynamics distillation. In *Advances in Neural Information Processing Systems*.

A Scientific Artifacts

The scientific artifacts used in the paper (e.g., LLaMa 2(7b), gpt-3.5-turbo, and Miniwob++) aligns with terms and conditions of usage of provided the original authors.

B Design choices

B.1 Impact of choice of LLM

It is important to acknowledge that while gpt-3.5-turbo shows superior performance as indicated in

Table 3, LLaMA-2 7B also demonstrates promising results in certain tasks, especially with one in-context example. This suggests that, despite some limitations like the current 8-bit quantization possibly affecting accuracy, there is potential for improved performance with larger LLaMA v2 models. This observation suggests the adaptability and potential effectiveness of our algorithm, even with smaller-scale models.

B.2 Impact of choice of Filtered VS Full DOM Elements

We discussed in our paper that we use only a subset of DOM Elements (referred to as 'Filtered DOM Elements') instead of the entire set of DOM Elements. We use the $getSummary()$ function to extract the filtered DOM elements. The $getSummary()$ function iterates through each DOM element and includes a DOM element in the filtered list if the element's tag or class belongs to a pre-defined list. In addition, we include the flags, an array of binary values. The values indicate whether a certain element has been clicked on/modified on. We display the pre-defined lists, called 'useful_tag' and 'useful_classes' below. The lists were determined experimentally.

```
useful_tag = {'button', 'text', 'input_time', 'textarea', '
    polygon', 'label', 'input_password', 'rect', 'tt', '
    circle', 'input_password', 'span', 'input_text', '
    input_number', 'input_date', 'input_radio', 'tspan', '
    input_checkbox', 't', 'button', 'h3', 'ul', 'a', 'p', '
    div', 'th','tr','td'}

useful_classes = {'folder'}
```

Listing 2: Predefined list of useful tags and classes

In Figure 8, we use a simple task to illustrate the differences between filtered DOM elements and full DOM elements.

B.3 Impact of choice of Sensitivity to prompts

In our analysis, we have observed that LLMs exhibit sensitivity to the prompts provided, including the system message and task message. We conducted a case study where we deliberately varied the task message and examined the resulting performance of the generated programs.

This sensitivity to prompts highlights the importance of carefully crafting and designing prompts to elicit the desired behavior and improve the overall performance of the language model.

```
task_message_1="""This task is a multi-step challenge. To
    successfully complete it, you need to be aware of the
    current state of the environment and the user input.
    Before performing any action, carefully observe and
    analyze the environment to determine whether further
    actions are required. When exploring and trying
    different actions, ensure that you select appropriate
    actions and arguments for the functions based on the
    current environment. Focus on efficiently reaching a
    solution by checking if the task can be solved with the
     current user input and environment state before taking
     any further steps, and by using correct actions and
    arguments for each function."""

task_message_2="""Next task is a multi-step task, directly
    performing a series of actions may not solve the task.
    Need to observe the changes in the user input before
    and after performing any action to see if
further actions need to be made to solve the task or not"""

task_message_3= """Next task is a multi-step task, directly
    performing a series of actions may not solve the task.
    Need to observe the changes in the user input before
    and after performing any action to see if
further actions need to be made to solve the task or not. So
    , explore and try different actions and figure out a
    way to solve the task, but at every step check if you
    are able to solve the task with the current user input
    before taking the action."""

task_message_4="""Your next task is a multi-step challenge.
    To successfully complete it, carefully observe and
    analyze the changes in user input before and after
    performing any action. This will help determine whether
     further actions are necessary. While it's important to
     explore and try various actions, always assess whether
     the task can be solved with the current user input
    before taking additional steps. Focus on efficiently
    reaching a solution without excessive exploration when
    a satisfactory outcome is already achievable.
"""

task_message_5="""The upcoming task is a multi-step
    challenge that requires you to pay close attention to
    the current user input. Your goal is to efficiently
    reach a solution by performing appropriate actions
    based on the present situation. Before taking any
    action, analyze the user input to determine if further
    actions are necessary. Explore and try the next action,
     but always ensure they are necessary, relevant to the
    current state and have the correct arguments for the
    functions. Continually assess the situation to check if
     the task can be solved with the current user input and
     environment state before proceeding further."""

task_message_6="""The upcoming task is a multi-step
    challenge. To successfully complete it, you must be
    aware of the current state of the environment and the
    user input. Before performing any action, carefully
    observe and analyze the environment to determine
    whether further actions are required. When selecting
    actions, ensure that you only perform actions if the
    current user input has the necessary elements. Focus on
     efficiently reaching a solution by trying to solve the
     task with the current user input and environment state
     before considering further exploration. Only explore
    and try different actions if the task cannot be solved
    with the current state. Make sure to use correct
    actions and arguments for each function based on the
    current environment."""

task_message_7="""The this task is a multi-step challenge.
    To successfully complete it, you must be aware of the
    current state of the environment and the objects in the
     image. Before performing any action, carefully observe
     and analyze the environment to determine whether
    further actions are required. When selecting actions,
    ensure that you only perform actions if the objects in
    the image have the necessary elements. Focus on
    efficiently reaching a solution by trying to solve the
    task with the current objects in the image and
    environment state before considering further
    exploration. Only explore and try different actions if
    the task cannot be solved with the current state. Make
    sure to use correct actions and arguments for each
    function based on the current environment."""

task_message_8="""The upcoming task is a multi-step
    challenge. Observe and analyze the environment and
    objects in the image before performing any action.
    Select actions based on the current state and ensure
    they are relevant to the objects in the image. Focus on
     solving the task with the current state, and only
    explore further if necessary. Use correct actions and
    arguments for each function, and be mindful of the
    environment during the process."""

task_message_9="""This task is a multi-step challenge.
    Observe and analyze the user input which contains the
    objects in the image before performing any action.
    Select actions based on the current state and ensure
    they are relevant to the objects in the image. Try
    other actions if and only if you are not able to solve
    the task with the current user input. Use correct
    actions and arguments for each function, and be mindful
     of the environment(user input) during the process."""
```

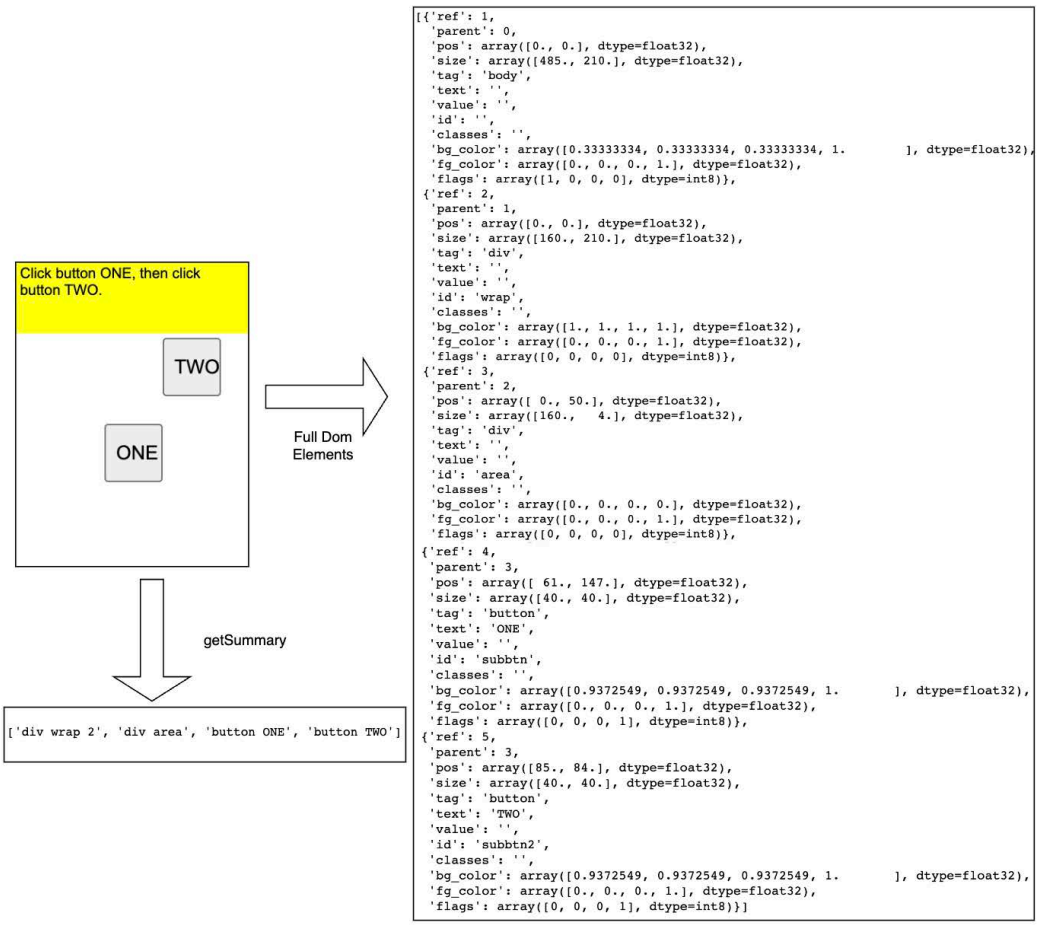

Figure 8: An example of comparing the full DOM elements VS filtered DOM elements for a simple task. We also include flags as part of the $getSummary()$ output.

```
task_message_10="""In this multi-step task, stay aware of
      the environment and user input. Observe and analyze
      before acting. As you try actions, choose suitable
      functions and arguments. Focus on efficiency: check if
      the task is solvable with current input and environment
       before proceeding. Converge toward the objective by
      using correct actions and arguments, and be cautious to
       avoid divergence."""

task_message_11="""This task involves a multi-step challenge
      , which can be accomplished by following these succinct
       steps:

1)Examine the environment by analyzing objects in the image
      from user input.
2)Determine if the desired element from the task image is
      present in the current objects.
3)If not, perform necessary actions (e.g., clicking,
      scrolling) to make the element available.
4)Iterate steps 1-3 until the desired element is found and
      can be clicked or interacted with.
5)Once the element is available and visible, execute the
      appropriate action on it."""


task_message_12="""
This task is a multi-step challenge, which can be
      accomplished by following these steps:
1)You should solve it step by step.
2)Before performing any action, determine if the desired
      element from the task is present in the Objects in
      Image.
3)If and only if the desired object is not there, say the
      phrase "The desired object is not there"
4)Then explore and perform other actions (e.g., clicking,
      scrolling) to see if the desired element is available
      in other states.
5)Iterate steps 2-4 until the desired element is found and
      can be clicked or interacted with.
"""

task_message_13="""
This task involves a multi-step challenge, which can be
      accomplished by following these succinct steps:
1)Determine if the desired element from the task image is
      present in the Object  in Image.
2)If its not, explore and perform other actions (e.g.,
      clicking, scrolling) to see if the element is available
       in other states.
3)Iterate steps 1-2 until the desired element is found and
      can be clicked or interacted with.
4)Once the element is available and visible, execute the
      appropriate action on it.
"""
```

Listing 3: Experiments on different task messages for API

The impact of variation in the task message on the performance of WebWISE, is shown in Table 5. Although the overall meaning of the "task message" remains consistent, minor differences in sentence structure and syntax can affect the model's performance for multi-steps tasks. While the average performance across tasks may appear similar, there are significant variations in accuracy among individual multi-step tasks, with some showing a high standard deviation of 62%. A future research direction is the development of methods robust to prompt variation.

B.4 Impact of Full DOM Elements

A deeper analysis of specific tasks like "Simple-Algebra" and "Navigate-Tree" revealed that providing the complete DOM elements as input improves performance. This improvement is attributed to the presence of keywords like "math-question" and "folder" within the DOM elements. Although these keywords may not be essential for executing actions within the web interface, they play a crucial role in providing contextual information to the language model. Consequently, the model gains a better understanding of the broader task it needs to accomplish at any given moment.

C Implementation Details

C.1 LLM Input

The prompt to the LLMs contains (in the following order) System message, API, Solution Description, Task description, and In-context examples. There are a total of 48 different tasks and each has its own in-context example. We provide the details for the system message and API below.

```
System_Message = """You are designed to generate programs to
      solve a wide range of complex web interface tasks.
You should be able to generate the program using either one
      or a composition of predefined action functions
along with general python codes to solve different tasks.
      You should not conversate with human in any context."""
```

Listing 4: First system message

```
Solution_Description="""Your task is to generate a solution
      for given problems based on objects in an image. Use
      the functions provided and follow these guidelines:

1)Construct solutions by calling functions and using Python
      data structures.
2)Solutions should be after the text 'Solution?'.
3)Only provide the function names without extra text in the
      solution.
4)Assume you can use observations without checking.
5)Don't assume additional functions or unknown information.
6)Add observation, reward, terminated, truncated, info = env
      .step(action) after each action.
7)Actions are independent of each other.
8)Do not add any comments, just return the code

If the task cannot be directly solved, perform a reasonable
      action and observe changes in the objects.
Use your DOM Elements knowledge to understand objects in the
      image.
Feel free to use Python constructs like if-else, for loop,
      while loop, etc., to generate the program."""
```

Listing 5: Second system message that appears after the API

```
task_message = """Your next task is a multi-step challenge.
      To successfully complete it, carefully observe and
      analyze the changes in user input before and after
      performing any action. This will help determine whether
       further actions are necessary. While it's important to
       explore and try various actions, always assess whether
       the task can be solved with the current user input
      before taking additional steps. Focus on efficiently
      reaching a solution without excessive exploration when
      a satisfactory outcome is already achievable. """
```

Listing 6: Third system message for multi-step methods

System Message The (first two) system messages were the same between tasks and across methods. A third system message is added for WebWise and Auto-Context Generation to ensure the task is completed step-by-step.

```
You should only use the functions provided herewith in the
      function description. Here is the list for the pre-
      defined functions [getSummary, click_action1,
      enter_text_action, scroll_action1].
To use a function, please refer to the Name, Input, Output,
      Description of the functions, and usage examples below.
       Action functions should be called correctly in the
      solution.

def getSummary(dom_elements):
    '''
    Input: DOM elements
    Output: Subset of DOM elements
    Description: get the filtered DOM elements from full DOM
       elements
    Example: objects_in_the_image = getSummary(dom_elements)
    '''
def click_action1(tag_class_name,id_text_name,observation):
    '''
    Input: tag or element, id or text, observation
    Output: clicks on specific element in environment
    Description: useful when you want to click on an element
       in the web interface. This function cannot be
      generalized on names. Normally first input is one of
      tag or element and second is test or id. The output is
      given as the action by calling click_action1 function
      or 'Cannot find in the DOM_element' if no such thing to
       be clicked on
    Example: Objects in Image: Button One;
             Task: Click button ONE;
             Solution: action = click_action1('button', 'ONE
      ', observation)
                       observation, reward, terminated,
      truncated, info = env.step(action)

    '''
def enter_text_action(input_text,observation):
    '''
    Input: text, observation
    Output: enters text in element in environment
    Description: useful when you want to type the input_text
       into input text box or a similar object like
      input_number that can accept text given the observation
       of the task interface. Need to call click_action1 to
      click on it before calling this function.
    Example: Objects in Image: input_text textbox;
             Task: Type 'Hello' into textbox;
             Solution: action = click_action1('input_text',
      'textbox', observation)
                       observation, reward, terminated,
      truncated, info = env.step(action)
                       action = enter_text_action('Hello',
      observation)
                       observation, reward, terminated,
      truncated, info = env.step(action)
    '''
def scroll_action1(text_to_scroll_to,observation):
    '''
    Input: text, observation
    Output: moves webpage such that certain text is visible
    Description: needed when elements do not appear on
       initial screen. Always used with other actions
    Example: Objects in Image: Button Apple
             Task: scroll and click button Apple
             Solution: action = scroll_action1('Apple',
      observation)
                       observation, reward, terminated,
      truncated, info = env.step(action)
                       action = click_action1('button','
      Apple',observation)
                       observation, reward, terminated,
      truncated, info = env.step(action)
    '''
```

Listing 7: Full API

Full API Below we list the full details our API. The API is constant between tasks and methods. For each function, we list the expected input, output, description and the example use case of the function. The example differs across the methods and depends on what visual information is provided. For the 'Instruction Only' method, there is

Tasks	Number of Functions	Incorrect Answers Visible (Y/N)	Target Button Not in Initial DOM (Y/N)
click-button-sequence	1	N	N
click-button	1	N	**Y**
click-checkboxes-large	Variable	**Y**	N
click-checkboxes-soft	Variable	**Y**	N
click-checkboxes-transfer	Variable	**Y**	N
click-checkbox	Variable	**Y**	N
click-collapsible-2	Variable	**Y**	**Y**
click-collapsible	2	N	N
click-dialog-2	1	**Y**	N
click-dialog	1	N	N
click-link	1	N	N
click-option	2	**Y**	N
click-tab-2-hard	Variable	**Y**	**Y**
click-tab2-easy	2	**Y**	N
click-tab2-medium	2	**Y**	N
click-tab-2	Variable	**Y**	**Y**
click-tab	1	N	N
click-test-transfer	1	**Y**	N
click-test-2	1	**Y**	N
click-test	1	N	N
click-widget	1	**Y**	N
enter-date	3	N	N
enter-password	3	N	N
enter-text-dynamic	3	N	N
enter-text-2	3	N	N
enter-text	3	N	N
enter-time	2	N	N
focus-text-2	3	**Y**	N
focus-text	3	N	N
guess-number	Variable	N	N
login-user	3	N	N
multi-layouts	1	N	N
use-autocomplete	1	N	N
grid-coordinate	1	N	N
simple-arithmetic	2	N	N
simple-algebra	2	N	N
navigate-tree	Variable	**Y**	**Y**
search-engine	Variable	**Y**	**Y**
find-word	Variable	**Y**	N
email-inbox-forward-nl-turk	3	**Y**	**Y**
email-inbox-forward-nl	3	**Y**	**Y**
email-inbox-nl-turk	Variable	**Y**	**Y**
email-inbox	Variable	**Y**	**Y**
terminal	Variable	**Y**	**Y**
click-pie	2	**Y**	**Y**
read-table	2	N	N
read-table-2	2	N	N
text-transform	1	N	N

Table 4: Classification of all tasks. 'Y' stands for yes. and 'N' stands for no.

no line that starts with 'Objects in Image' since that information is not part of the method.

C.2 Scatter Plots

We display additional scatter plots for gpt-3.5-turbo results in Figures 9, 10, and 11.

D Task Analysis

D.1 Task Classification

In the results section, we categorized the tasks based on the number of pre-defined functions needed. However, we also introduce two alternative methods for classifying the tasks. We provide a comprehensive table, Table 4, that presents all the tasks along with their respective classifications.

Incorrect Answers Present One way we can classify the tasks is, if there are incorrect answers present. Tasks that include incorrect answers are characterized by the presence of multiple clickable buttons, as opposed to tasks with a single button. An illustration showcasing a task with and without incorrect answers is provided in Figure 13.

Target Button not in Initial DOM Elements A second way we can classify tasks is based on the

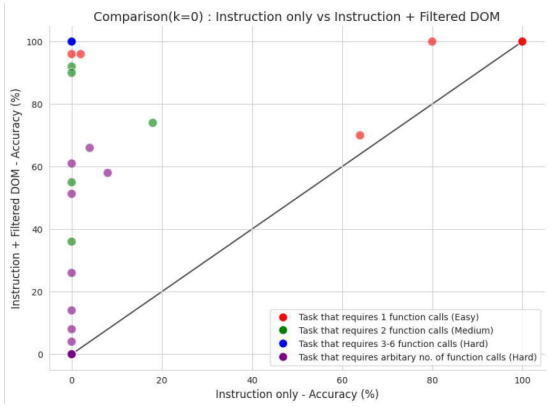

Figure 9: Comparison of k=0 performance across all tasks for Instruction Only (x-axis) and Instruction+Filtered DOM (y-axis)

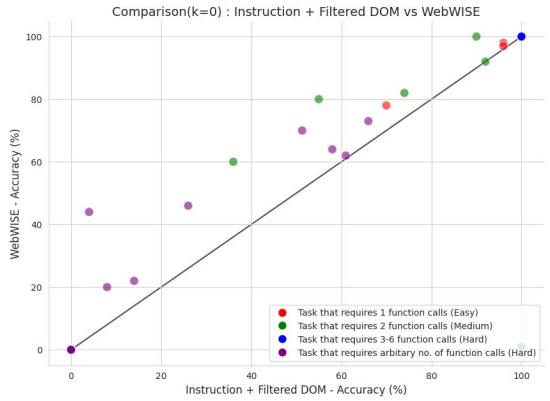

Figure 10: Comparison of k=0 performance across all tasks for Instruction+Filtered DOM (x-axis) and WebWISE (y-axis).

initial set of DOM elements. In simpler tasks, the initial set of DOM elements already provides all the necessary information to perform the task. However, for more complex multi-step tasks, it is required to perform at least one additional step to access the DOM elements that contain the target button or the information needed to execute the task correctly. Typically, this additional step involves clicking on a button that triggers a screen change or reveals the relevant elements. Figure 12 provides an example illustrating this concept.

D.2 Task Failures

In this section, we explore why certain tasks failed. It is worth noting that some failures are attributed to the different task classifications we discussed earlier.

One specific example is the 'click-dialog' task, as demonstrated in Table 1. When using our WebWISE method with only one example, we

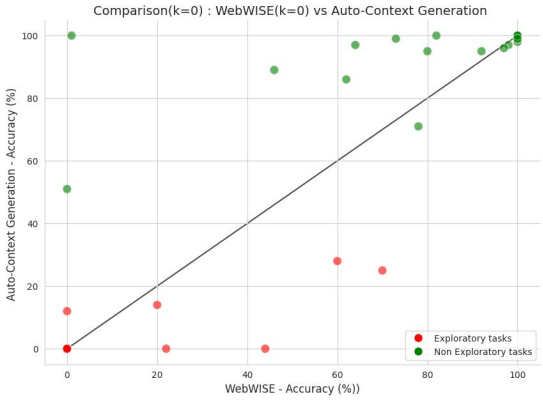

Figure 11: Comparison of WebWISE (x-axis) vs. Auto-Context Generation (ACG) (y-axis) at k=0 across various tasks. Orange dots indicate tasks requiring exploration, such as those where the target button isn't present initially in the DOM elements, whereas green dots represent non-exploratory tasks. A noticeable trend is that Auto-Context Generation's (ACG's) performance decreases on exploratory tasks. When ACG finds a correct solution during the zero-shot trials, it will continue to use the same solution found instead of generalizing from it.

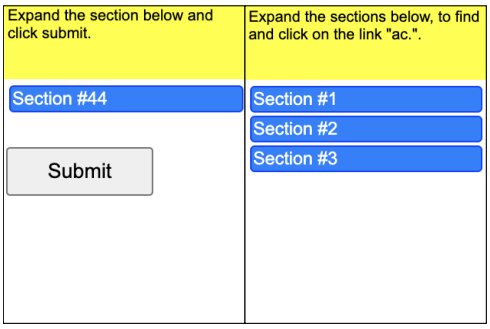

Figure 12: Comparison of tasks where target button is visible and not visible in the initial set of DOM elements. The target is visible for task shown on the left image and initial DOM elements contain all relevant information. For task shown on the right image, to get the relevant information, different sections have to be expanded, and each time a section is expanded, the DOM elements are changed.

achieve perfect accuracy. However, for the 'click−dialog−2' task, the accuracy drops to 76%. This discrepancy can be attributed to a particular sub-task where the task description instructs the user to click on the 'x' symbol to close a dialog. Language models may not fully comprehend that the symbol 'x' represents the close function. As a result, this lack of understanding leads to failures in executing this specific sub-task.

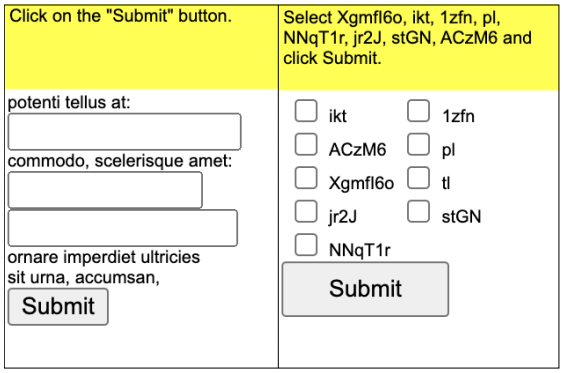

Figure 13: Comparison of tasks with and without incorrect answers. For image on the left, incorrect answers are NOT present. There is only a single button to click. However, for image on the right, ncorrect answers are present. Eg. only some of the checkboxes should be clicked on.

Email Tasks Among the tasks we evaluated, some simulate performing actions in an email mobile app. Performance varies widely across these tasks, particularly those that begin with the word "email." Tasks starting with "email-inbox-forward" tend to have better performance compared to tasks without the word "forward". This variation can be explained by the scope of the individual tasks. In tasks with "forward" in the prompt, there is only one specific action that needs to be performed: forwarding an email. However, in tasks without the word "forward," there can be one of four possible actions: forwarding, starring (marking as important), deleting, and replying to an email. To maintain consistency with the other tasks, we used a single in-context example with only one of the actions for the email-related tasks. Consequently, our method can only successfully execute the action mentioned in the prompt.

Terminal One common source of failure in tasks involving a terminal is the model's lack of knowledge on how to execute a command by pressing the enter key. The task only succeeds when we provide an example because we pass the first argument to $enter\_text\_action$ as "$CommandToBeEntered\backslash n$", which simulates pressing the enter key after entering the command. To address this issue, a potential solution in the zero-shot scenario is to develop primitive functions specifically for key actions such as "ENTER," "BACKSPACE," or "DELETE." By incorporating these primitive functions into our models, it can learn to perform key-related actions more effectively in terminal-based tasks.

Search Engine and Text Transform Failures in tasks like "Search-Engine" and "Text-transform" can be attributed to the limitations of the filtered DOM elements. For example, in the search engine task, the instruction may involve clicking on the 8th search result on a webpage. While the full DOM elements contain the search results in the correct order, the filtered DOM elements do not. A similar observation can be made for tasks like "Text Transform." To address this issue, it becomes necessary to develop an adaptive $getSummary()$ function that can extract the most relevant elements while also preserving their order within the DOM. Alternatively, approaches involving the use of image input modalities could be explored to overcome these limitations. Such approaches can provide a visual representation of the webpage, enabling the model to better understand the layout and order of the elements present.

D.3 Additional Task Analysis

In tasks like "Copy-Paste," the objective is to copy text from the task interface and paste it into an empty text field, as illustrated in Figure 14. However, we did not implement the copy-paste function for the LLMs to interact with the environment and complete the task. We noticed that LLMs make references to functions like "$create\_copy\_action$" and "$create\_paste\_action$," which, if implemented, could have led to the correct solution (as shown in Figures 15,16). Some additional functions such as clicking on specific coordinates could also be implemented to improve the LLMs' capabilities in handling tasks like these.

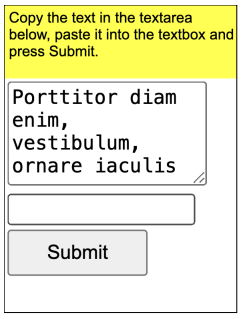

Figure 14: An example of the copy-paste task

D.4 Single step approach diagram

The example of single step approach is shown in the Figure 17.

Various Task Message	**Average performance on WebWise k=0**	**Average performance on WebWise k=1**
Task Message 1	0.62	0.85
Task Message 2	0.52	0.72
Task Message 3	0.50	0.76
Task Message 4	0.54	0.74
Task Message 5	0.48	0.74
Task Message 6	0.54	0.76
Task Message 7	0.56	0.76
Task Message 8	0.56	0.78
Task Message 9	0.58	0.78
Task Message 10	0.60	0.80
Task Message 11	0.56	0.76
Task Message 12	0.54	0.78
Task Message 13	0.58	0.80

Table 5: WebWISE k=0 and k=1 performance for the different task messages which indicates the sensitivity of the task message

Tasks	WebWISE (k=1)	RCI	RCI k	Result
click-button-sequence	**1.00**	**1.00**	2	Draw
click-button	**1.00**	**1.00**	1	Draw
click-checkboxes-large	**1.00**	0.94	1	Win
click-checkboxes-soft	**0.78**	0.72	1	Win
click-checkboxes-transfer	**1.00**	**1.00**	2	Draw
click-checkbox	**1.00**	**1.00**	2	Draw
click-collapsible-2	**0.66**	0.62	2	Win
click-collapsible	**1.00**	**1.00**	1	Draw
click-dialog-2	0.74	**1.00**	3	Lost
click-dialog	**1.00**	**1.00**	1	Draw
click-link	**1.00**	**1.00**	n/a	Draw
click-option	**1.00**	**1.00**	1	Draw
click-tab-2-hard	0.68	**0.76**	6	Lost
click-tab-2	**0.78**	0.74	1	Win
click-tab	**1.00**	**1.00**	1	Draw
click-test-2	**1.00**	**1.00**	n/a	Draw
click-test	**1.00**	**1.00**	1	Draw
click-widget	**0.98**	**0.98**	3	Draw
enter-date	**1.00**	0.96	4	Win
enter-password	**1.00**	**1.00**	n/a	Draw
enter-text-dynamic	**1.00**	**1.00**	n/a	Draw
enter-text	**1.00**	**1.00**	n/s	Draw
enter-time	**1.00**	**1.00**	2	Draw
focus-text-2	**1.00**	**1.00**	n/a	Draw
focus-text	**1.00**	**1.00**	1	Draw
guess-number	**0.84**	0.20	n/a	Win
login-user	**1.00**	**1.00**	n/a	Draw
multi-layouts	**0.78**	0.72	n/a	Win
use-autocomplete	**0.92**	0.58	n/a	Win
grid-coordinate	**1.00**	**1.00**	1	Draw
simple-algebra	**1.00**	**1.00**	1	Draw
navigate-tree	**0.92**	0.86	1	Win
search-engine	0.20	**1.00**	22	Lost
email-inbox-forward-nl-turk	0.92	**0.94**	3	Lost
email-inbox-forward-nl	0.96	**1.00**	3	Lost
email-inbox-nl-turk	0.18	**0.98**	4	Lost
email-inbox	0.21	**0.98**	6	Lost
terminal	0.93	**1.00**	1	Lost
Average Performance	0.88	0.92		

Table 6: Comparison of WebWISE(k=1) and RCI, both using gpt-3.5-turbo. The second last column lists the k value for RCI if available. Although the average performance of the RCI method is marginally higher than our approach, it is to be noted that the number of examples provided the RCI method are comparatively higher as seen in the table. RCI also has higher performance for some tasks when using gpt-4, which is also likely to be in our case since gpt-4 is a more powerful LLM. The "Result" column shows whether our method, "WebWISE (k=1)," was successful or not compared to "RCI (k=n)."

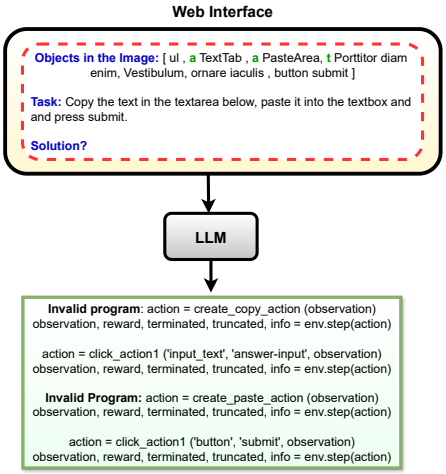

Figure 15: The actual generated program for the copy-paste task in Zero Shot scenario where we observe "$create\_copy\_action$" being created by the model even though that function was not provided in the model API

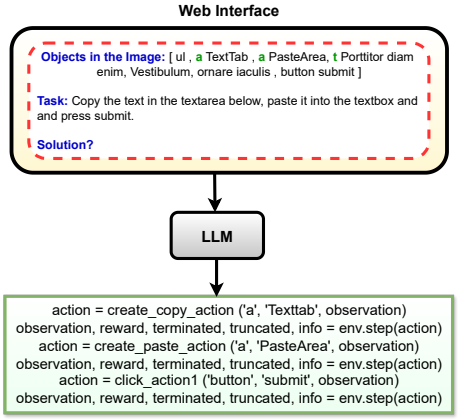

Figure 16: A potential generated program for the copy-paste task in Zero Shot scenario where the model could create functions like "$create\_copy\_action$" and "$create\_paste\_action$" to solve the task.

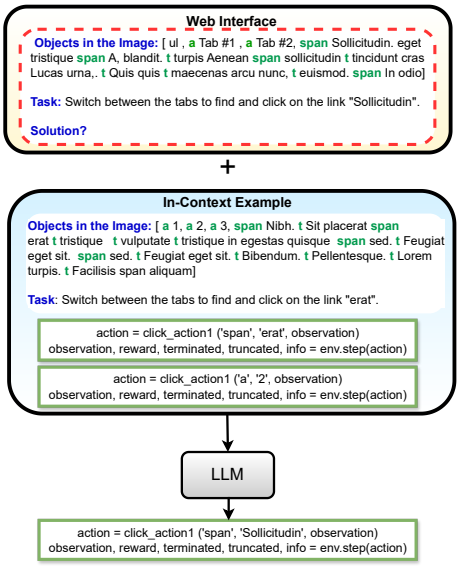

Figure 17: An example of the single-step approach. The input to LLMs includes the in-context example (in the blue box) and the task query along with the filtered DOM elements(in the orange box).