# The Unsolved Challenges of LLMs as Generalist Web Agents: A Case Study

**Rim Assouel**[*†] **Tom Marty**[*†] **Massimo Caccia**[‡] **Issam Laradji**[‡] **Alexandre Drouin**[‡§]

**Sai Rajeswar Mudumba**[‡] **Hector Palacios** [‡] **Quentin Cappart** [¶] **David Vazquez**[‡]

**Nicolas Chapados**[‡§] **Maxime Gasse**[*‡§] **Alexandre Lacoste**[*§]

## Abstract

In this work, we investigate the challenges associated with developing goal-driven AI agents capable of performing novel tasks in a web environment using zero-shot learning. Our primary focus is on harnessing the capabilities of large language models (LLMs) as generalist web agents interacting with HTML-based user interfaces (UIs). We evaluate the MiniWoB benchmark and show that it is a suitable yet challenging platform for assessing an agent's ability to comprehend and solve tasks without prior human demonstrations. Our main contribution encompasses a set of extensive experiments where we compare and contrast various agent design considerations, such as action space, observation space, and the choice of LLM, with the aim of shedding light on the bottlenecks and limitations of LLM-based zero-shot learning in this domain, in order to foster research endeavours in this area. In our empirical analysis, we find that: (1) the effectiveness of the different action spaces are notably dependent on the specific LLM used; (2) open-source LLMs hold their own as competitive generalist web agents when compared to their proprietary counterparts; and (3) using an accessibility-based representation for web pages, despite resulting in some performance loss, emerges as a cost-effective strategy, particularly as web page sizes increase.

## 1 Introduction

One long-sought challenge of AI is to develop goal-driven agents that can accomplish novel tasks in a zero-shot fashion by simply describing a goal to the agent. The field of reinforcement learning aims to achieve this using meta-learning and goal-conditioned RL [Oh et al., 2017, Laskin et al., 2023]. However, the lack of proper large-scale datasets has hindered progress in this direction. On the other end, recent development in large language models (LLMs) has shown an unprecedented potential to leverage language fluency, common sense and strong coding capabilities to make giant leaps in the direction of zero-shot goal-conditioned agents [Liang et al., 2022, Li et al., 2022, Carta et al., 2023, Du et al., 2023, Wang et al., 2023]. Specifically, recent work shows how to leverage LLMs to navigate an HTML-based user interface (UI) by prompting the model with the goal and HTML content. The LLM then generates text actions that are executed through backend code [Kim et al., 2023, Gur et al.,

[*]These authors contributed equally to this work.

[†]Mila Québec. Work done while interning at ServiceNow. Correspondence to: assouelr@mila.quebec and tom.marty974@gmail.com

[‡]ServiceNow Research. Correspondence to: All authors <firstname.lastname@servicenow.com>.

[§]Mila Québec

[¶]Polytechnique Montréal. Correspondence to: quentin.cappart@polymtl.ca

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

**Table 1:** *An overview of existing web agents for MiniWoB. 'BC' and 'RL' respectively stand for Behavior Cloning and Reinforcement Learning. Notably, the three agents listed below the horizontal line were not fine-tuned for MiniWoB tasks.*

| Agent | Method | Input | Output | Success rate | Task subset |
|---|---|---|---|---|---|
| DOMNet [Liu et al., 2018] | RL | html | high-level (elem id) | 81% | 48 |
| CC-Net [Humphreys et al., 2022] | BC + RL | pixels + html | low-level (x, y) | 94% | 104 |
| WebN-T5 [Gur et al., 2023b] | pre-trained + BC | html | high-level (elem id) | 51% | 48 |
| WebGUM [Furuta et al., 2023] | pre-trained + BC | pixels + html | low-level (x, y) | 80% | 56 |
| Pix2Act [Shaw et al., 2023] | pre-trained + BC + RL | pixels | low-level (x, y) | 96% | 59 |
| RCI [Kim et al., 2023] | pre-trained + few-shot + prompting | html | high-level (XPath) | 94% | 47 |
| SYNAPSE [Zheng et al., 2023] | pre-trained + few-shot | html | high-level (XPath) | 98% | 63 |
| WebAgent [Gur et al., 2023a] | pre-trained + decomposition | html | code | 76% | 56 |

2023a, Iki and Aizawa, 2022]. While benchmarks like MiniWoB [Shi et al., 2017, Liu et al., 2018] offer a critical evaluation platform, current top-performing algorithms often rely on exhaustive human demonstrations or numerous offline task interactions, underscoring a crucial gap in achieving novel tasks accomplishment in web environments [Humphreys et al., 2022, Furuta et al., 2023, Yao et al., 2022].

In this work, we re-examine MiniWoB through a lens focused on zero-shot learning to give it a fresh start. We show that it makes for a challenging platform to test a generalist agent's ability to understand and solve tasks without prior human demonstration or offline interaction with the task. Our main contribution is an extensive set of experiments that explore various bottlenecks and limitations. Specifically, we aim to understand the effect of various agent design choices by addressing the following questions:

**Action Space** Should LLM agents be confined to a small set of low-level actions (`click`, `type`) or can they benefit from a more flexible action space, such as code?

**Observation Space** Should LLMs better process a concise summary of pages (e.g., an accessibility tree, as in Zhou et al., 2023) or work directly with the raw HTML?

**Choice of LLM** Are closed-source LLMs like GPT-4 [OpenAI, 2023] necessary to achieve zero-shot generalization or can recent open-source models achieve comparable performances?

By shedding further light on these questions, we aim to offer comprehensive insights that are valuable for further research in generalist web agents. Our empirical results indicate that: (1) the effectiveness of the code-based action space is notably dependent on the specific LLM used; (2) open-source LLMs are competitively viable generalist web agents compared to closed-source alternatives; and (3) an accessibility-tree-based web page representation, although slightly less effective, becomes increasingly cost-efficient as web pages grow in size.

## 2 Related works

**Benchmarks.** Evaluating the performance of web agents on novel tasks is a challenge in itself, and several benchmarks have been proposed over the years. MiniWoB [Shi et al., 2017, Liu et al., 2018] is a generic collection of 125 heterogeneous web tasks that range from clicking a specific button in a form to using a basic text editor or an email box. While it has been essentially solved using RL [Humphreys et al., 2022], it remains a benchmark of choice for generalist web agents (see an overview of MiniWoB web agents in table 1). Other recent benchmarks include WebShop [Yao et al., 2022], a simulated e-commerce website with shopping tasks, and WebArena [Zhou et al., 2023], a collection of heterogeneous tasks on realistic websites that emulate real-world domains.

**Fine-tuned web agents.** Liu et al. [2018] propose DOMNet, an attention-based neural network trained via reinforcement learning (RL) on MiniWoB. Humphreys et al. [2022] propose CC-Net, a multimodal transformer trained via behavioural cloning (BC) and RL on MiniWoB. Gur et al. [2023b] propose WebN-T5, a pre-trained T5 model fine-tuned via BC on MiniWoB. Furuta et al. [2023] propose WebGUM, a multimodal transformer that combines a pre-trained Flan-T5 with a ViT vision model and is fine-tuned via BC on MiniWoB. Shaw et al. [2023] propose Pix2Act, an image-to-text architecture that combines a pre-trained ViT vision model and a T5 transformer, fine-tuned via BC and RL (using Monte-Carlo tree search) on MiniWoB.

**Foundation web agents.** Liu et al. [2023a], Liu et al. [2023b] and Yao et al. [2023] report the performance of pre-trained LLMs as web agents on the WebShop benchmark. They use as input the HTML and output high-level actions (`search[query]`, `click[elem]`). These studies highlight a large performance gap between closed-source LLMs (e.g., OpenAI GPT models) and open-source ones. A tendency for the strongest models to be less sensitive to the prompting strategy is also observed. Kim et al. [2023] propose to *Recursively Criticize and Improve* (RCI), a self-correcting prompting scheme for LLMs, and report the performance of a web agent built upon GPT-3.5 on MiniWoB. Zheng et al. [2023] propose SYNAPSE, a few-shot prompting strategy for GPT-3.5, which recovers relevant demonstrations dynamically for each new task by querying a database of examples built using task embeddings. Gur et al. [2023a] propose WebAgent, tailored for real-world websites with large HTML. It relies on a pre-trained HTML-T5 to decompose the HTML into sub-task snippets and a pre-trained Flan-U-PaLM to generate Selenium[6] Python code for each sub-task.

While a broad range of web agent solutions have been proposed in the literature, the problem of building generalist web agents is far from solved. Even on the MiniWoB benchmark—which consists of short-horizon, toy tasks—state-of-the-art solutions require either fine-tuning with human demonstration [Zheng et al., 2023, Shaw et al., 2023], RL interactions [Humphreys et al., 2022], or a large collection of demonstrations and few-shot exemplars [Zheng et al., 2023]. In addition, most MiniWoB studies only consider a filtered subset of tasks for evaluation (see table 1) and avoid use-cases in which long HTML context, complex interaction patterns, or task complexity could be an issue.

## 3 Challenges of the World Wide Web

Performing tasks in the web poses a specific set of challenges for LLM-based web agents. Websites are often not designed for automated navigation using algorithms. Web developers commonly integrate security measures like CAPTCHA or code obfuscation to prevent malicious algorithms from impersonating a user or executing repetitive tasks with ease. The heterogeneity and the wilderness of web security standards (iframes, shadowDOMs), browsers (Chrome, Edge, Safari, Firefox, etc.), and embedded technologies (media players, plugins) introduce many challenges, with website implementations ranging from static HTML pages (web 1.0) to responsive JavaScript widgets and full-blown smartphone applications (web 2.0).

**Understanding web UIs.** A webpage is typically represented as single HTML document, the HTML DOM tree [World Wide Web Consortium, 2004], which represents UI elements (titles, lists, buttons, icons) and is rendered as 2D image by the web browser. A straightforward solution for LLM agents is to process the HTML directly, and assume that it gives sufficient understanding of the webpage for interacting with it. This solution, while simple, suffers from several limitations:

- Context length: on real-world websites the HTML document can be lengthy and verbose, and go beyond the maximum context length of currently available LLMs. An ad-hoc solution is to prune the HTML of unimportant elements and attributes, at the risk of removing important information. Another option, used in WebArena [Zhou et al., 2023], is to rely on the so-called accessibility tree[7] offered by the Chrome browser, which aims at offering a short textual summary of every element on the webpage for visually impaired people, as an effort to support the Accessible Rich Internet Applications[8] (ARIA) standard.
- Visual information: the HTML document by itself is not sufficient to render the UI of a webpage. Style information, which dictate how the webpage is rendered, is typically delegated to a different set of files (CSS files). Some elements have a stochastic rendering, and require to load third-party snippets (advertisements), while some UIs are entirely dynamic.[9] Due to the lack of standards in implementations, a universal text representation would need to embed not only the list of elements present on the webpage, but also their size, their position, whether they overlap, etc. In MiniWoB, several tasks require a precise 2D understanding of the webpage (see fig. 1).
- Domain-specific behaviors: another challenge related to the wilderness of the web, is the heterogeneity of UI implementations and behaviors (e.g., autocomplete rules, drag-and-drop). Touchscreens

[6] Selenium is a Python API for automated browser interaction, see https://www.selenium.dev/.
[7] https://developer.chrome.com/docs/devtools/accessibility/reference/
[8] https://www.w3.org/TR/wai-aria/
[9] E.g., jQueryUI, https://jqueryui.com/

make this problem worse, with increasing numbers of webpages behaving like responsive smartphone applications, while legacy, static webpages are still common among public entities.

**Interacting with web UIs.** Another challenge for LLMs is to act in a browser environment. While sending low-level commands (click coordinates, press key) might seem expressive enough to emulate the behaviour of any human user, it might not be the best design choice for an generalist LLM web agent. A reason for this is that low-level commands might be misaligned with the LLM input, which might not contain the precise coordinates of each element in the UI, as explained earlier. But another reason also is that generalist LLMs trained on text might be more aligned with high-level XPath commands such as `click input[type=submit]` rather than `click 32.5, 78.4`. Another option is to interact with the browser directly through code, for example by producing Selenium instructions in Python. An advantage is that generalist LLMs might be well-aligned if trained on code, in particular Selenium code. A potential disadvantage is that through Selenium, one can perform not only mouse and keyboard commands, but also Javascript code execution in the browser. This gives LLM agents the option to bypass the UI entirely, which might not be desirable[10] and could introduce additional security vulnerabilities such as Self Cross Site Scripting (XSS). Lastly, another limitation of LLMs when interacting with webpages is the delay in obtaining the next action, which might be incompatible with the dynamic and responsive nature of some websites.

**Web task complexity.** Solving novel tasks on the web requires a large set of cognitive abilities. For instance, on-the-fly adaptation with exploration and self-correction is crucial to face the diversity of the web. Multi-step reasoning, planning and memorization are also necessary for a wide variety of tasks involving search, comparison, and decision-making (e.g., searching information on a website, then filling a form on another website).

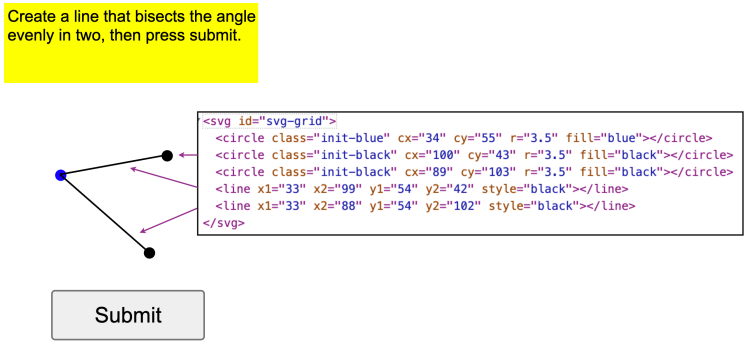


**Figure 1:** *Solving the MiniWoB task called* `bisect-angle` *only using the HTML code of the webpage requires 2D understanding and reasoning on the rendered scene.*

## 4 Agent design

When designing language model-based web agents, it is crucial to consider various factors that can significantly impact their effectiveness and performance. In our research, we have identified three primary sources of bottlenecks in the design of LLM-based web agents: action space, observation space, and backend LLM. Each presents distinct challenges and opportunities for optimizing agent performance. This section explores these bottleneck sources and their implications for tasks in the MiniWoB domain.

**Observation Space.** The observation space determines how the web agent perceives and understands webpages. We investigate two primary approaches:

- **HTML**: This approach directly parses raw HTML content, offering an unfiltered view of the webpage. While this might be beneficial for comprehending complex webpages, it can result in a long and verbose context for the LLM, which may demand more computational resources to process. In our experiment we opt for a light pruning of the HTML, where we only remove `<style>`, `<script>` and `<link>` tags, as well as HTML comments.
- **Accessibility Tree**: In contrast, accessibility-based observation streamlines data extraction, potentially reducing context length and enhancing efficiency. However, the custom format of

[10] For example, the Selenium action `driver.execute_script('core.endEpisode(1)');` will terminate the episode with reward 1 in MiniWoB.

accessibility data may not align with what the LLM has encountered during training. This mismatch may necessitate additional training support to bridge the gap between the custom observation format and the LLM's understanding of web content.

The choice between these observation paradigms affects the trade-off between context length, data extraction efficiency, adaptability, and training effort. Note that in our experiments we further augment both HTML and accessibility tree elements with a backend ID (`bid`) attribute in order to identify elements unambigously, as well as their (`x, y`) coordinates on the screen in order to give a glimpse of the visual rendering of the webpage to the LLM.

**Action Space.** The action space defines the set of actions available to an LLM-based web agent for interacting with webpages. We categorize the action space into three levels:

- **High-Level Actions**: These actions directly target HTML elements using unique high-level identifiers. E.g., `{'action': 'click', 'bid': '103' }`.
- **Low-Level Actions**: Encompassing granular actions like simulating mouse events or directly manipulating HTML coordinates, low-level actions provide fine-grained control but can be complex to use effectively. E.g., `{'action': 'click', 'x': '42.2', 'y': '42.2'}`.
- **Selenium**: Pure code actions are defined using code snippets, specifically restricted to Selenium code in this study. E.g., `driver.find_element(...).click()`.

Incorporating pure code actions allows the agent to select complex, high-level actions, potentially surpassing the designer's explicit definitions. This adaptability empowers the agent to compose actions, simplifying planning and enhancing performance across diverse scenarios. We describe in detail the different action spaces considered in our experiments in section A.2

**Action Format.** Within actions that do not correspond to Selenium code output, we distinguish between two formatted output types that correspond to the output expected from the LLM at each time step when solving a task :

- **Single** action in a JSON format. E.g., `{'action': 'click', 'bid': '103' }`.
- **Multiple** actions in a list of JSON actions : `[{'action': 'click', 'bid': '103' }, {'action': 'type', 'bid': '83', 'text': 'Hello!'}]`.

Allowing multiple actions in a single LLM call can enhance computational efficiency by reducing the number of calls required.[11] However, this efficiency comes with a trade-off. Unexpected behaviors on the webpage, like pop-up windows, can prematurely end an episode when multiple actions are executed together. In contrast, using a single action per LLM call provides greater adaptability but may involve more calls. This trade-off between single and multiple actions per LLM call reflects a critical consideration in agent design and performance optimization.

**LLM Model.** Choosing the backend Large Language Model (LLM) is a critical agent design decision with several pivotal factors:

- **Open vs Closed Source**: Open-source LLMs like LLAMA [Touvron et al., 2023a] and Falcon [Penedo et al., 2023] are great for research. Commercial ones like GPT excel in performance and are optimized for production. The choice depends on cost, speed, privacy, and use case.
- **Model Size**: Larger models generalize better but require more computational power and may increase the latency of response.
- **Context Length**: LLMs have fixed token limits, posing a challenge when dealing with real-world HTML files that can easily surpass the limits of common context windows. Open-source models offer the flexibility of fine-tuning to accommodate larger context sizes.
- **Fine-Tuning**: Though not tested here, fine-tuning LLMs for specific tasks can improve performance and allow for domain-specific knowledge integration.

In our study, we keep the agent design fixed and switch the backend LLM between 4 choices. GPT-3.5, GPT-4, StarChat $\beta$ [Tunstall et al., 2023] and Code Llama [Rozière et al., 2023].

[11] This is particularly convenient in tasks like *use-spinner* where the agent must click $n$ times on the spinner buttons to enter the number $n$. When the number is 10, it takes 11 steps to solve the task using single actions.

# 5 Empirical Study

In this section, we begin by introducing sub-benchmark categories to identify current limitations in LLM-based web agents. We then conduct experiments to investigate how various design factors influence the performance of these generalist agents. Specifically, we concentrate on three main elements: the action space (section 5.3), the Large Language Model (section 5.4), and the observation space (section 5.5).

**Agent Naming.** The naming convention for the agents we consider in the experiments follows a consistent pattern: `observation_actionformat_actionspace-llm`. In this format, `observation` refers to the type of representation the agent works with, `actionformat` specifies the expected output format from the language model, `actionspace` defines the range of actions it can perform (e.g., high, low-level, or code actions), and `llm` indicates the underlying language model used by the agent. For example, `html_multi_high-gpt-3.5` signifies an agent designed for HTML data, outputting a list of multiple high-level actions at every LLM call, and powered by the GPT-3.5 language model.

## 5.1 Experimental setup

We ran our agents on 125 MiniWoB tasks and reported results on sub-benchmarks as described in Section 5.2. Each agent can interact for up to 10 steps per episode, which is sufficient for most environments. Each task is repeated 10 times with different seeds. Since LLMs are generally slow and cannot solve tasks in real time, we ignore time penalties that are normally provided by MiniWoB. Specifically, we care about task success only and we report a reward of 1 for any positive reward, 0 otherwise, as is common in the literature.[12] Statistical uncertainties are reported using a stratified bootstrap[13] of cumulative rewards over the 10 seeds and averaged across all tasks of a benchmark.

## 5.2 Sub-benchmark Analysis

For the purpose of our analysis, we propose a taxonomy of sub-benchmarks for the complete set of 125 web tasks proposed in MiniWoB. This comprehensive classification illustrates the diverse challenges embedded in each task, thereby offering a balanced analysis and understanding of how large language models manage these complexities. The eight overlapping sub-benchmarks are laid down in detail in fig. 5 of the Supplementary Material:

- **Easy (25)**: tasks such that 50% of our agents had an success rate above 80%.
- **Hard (18)**: tasks such that 90% of our agents had a success rate below 10%.
- **WebGUM (56)**: a filtered subset of 56 tasks used by Furuta et al. [2023] and Gur et al. [2023a].
- **Long Context (11)**: based on the length of the pruned HTML of the webpage, long context tasks involve extensive contextual information, demanding the model's ability to comprehend and process lengthy sequences of data. For example, the task `click-pie` involves intricate SVG graphics.
- **Pixel (12)**: tasks requiring an understanding of the visual rendering of the webpage; e.g. `count-sides` requires counting the number of sides of a rendered polygon.
- **2D Understanding (30)**: tasks requiring two-dimensional understanding and reasoning, and/or precise coordinate-based actions (`click x, y`), emphasizing spatial understanding; e.g. `bisect-angle` requires computing a bisector angle based on coordinates of multiples points and lines.
- **Domain-Specific Knowledge (46)**: tasks that demand domain-specific knowledge, which one can only obtain from prior interaction or through few-shot examples (task-specific prompting); e.g. `enter-date` requires understanding how to interact with the native HTML element `<input type="date">`, which can vary depending on the browser being used. We illustrate this category with visual examples 8 and 9 in the Supplementary Material.
- **Long Episode (8)**: tasks that involve extended sequences of actions ($> 10$), requiring the model to maintain coherence and strategy over time. Agents capable of executing more than one action at a time (e.g., with code) should be advantaged in such environments; e.g. `choose-date` requires to browse the months of the year in a custom widget.

[12] Certain limitations should be noted regarding this metric, as in certain MiniWoB tasks achieving a strictly positive reward may not necessarily indicate the completion of the task.

[13] We sample 10 times with replacement the reward of the 10 repeated episodes, and average across tasks.

- **Action Space Limited (9)**: tasks where agents would greatly benefit from an action space richer than (click, type), such as code or more high-level actions (e.g., drag-and-drop). For example, the `drag-items` task.
- **Requires Augmented HTML (14)**: tasks that may require additional information that is not originally listed in the raw source code of the webpage; e.g. in the task `sign-agreement`, the agent needs to scroll inside a `div`. In this case, enhancing the HTML with an attribute indicating whether it is possible to scroll inside the  would be particularly useful.

Overall, our results indicate that the sub-categories we introduced are considerably more challenging than the traditionally reported subsets (see table 1), thereby highlighting the unresolved complexities inherent to the development of generalist web agents. In Appendix A.1, we provide a set of representative examples illustrating some of the challenges listed above.

### 5.3 Impact of the Action Space

In this subsection, we narrow the observation space to HTML and focus exclusively on GPT models to assess the influence of action space choices. The results are depicted in Figure 2.

Focusing initially on GPT-3.5, we found that high-level actions are usually better than low-level ones, and only rarely does allowing both hinder performance. Notably, we observe that code actions, in particular, outperformed high-level actions when using GPT-3.5 when considering all the tasks. Two plausible explanations for this phenomenon are: (1) the inherent flexibility of code actions, which can simplify the planning process by enabling the agent to dynamically compose actions and/or use more complex actions that were not anticipated in the backend by the action space designer (e.g. drag-and-drop action); (2) Custom high-level actions, on the other hand, maybe more prone to being out of distribution given the specific training data the LLM has encountered.

As expected, when transitioning from GPT-3.5 to GPT-4, we observed an overall improvement in performance. This transition seems to bridge the gap between the code action format and high-level actions. It suggests that a more advanced LLM, such as GPT-4, enhances the agent's understanding of the intended actions, potentially making custom high-level actions more effective. This demonstrates the dynamic interplay between LLM capability and action format choices in influencing agent performance.

Although performing multiple actions per time step generally leads to improved results on average, we observe instances where agents executing multiple actions become overly confident about the website's stateliness. This can lead to situations where an action that has drastically changed the website state rendered subsequent planned actions invalid. We believe that this issue could be effectively addressed through prompting, by adding warnings, and error feedback.

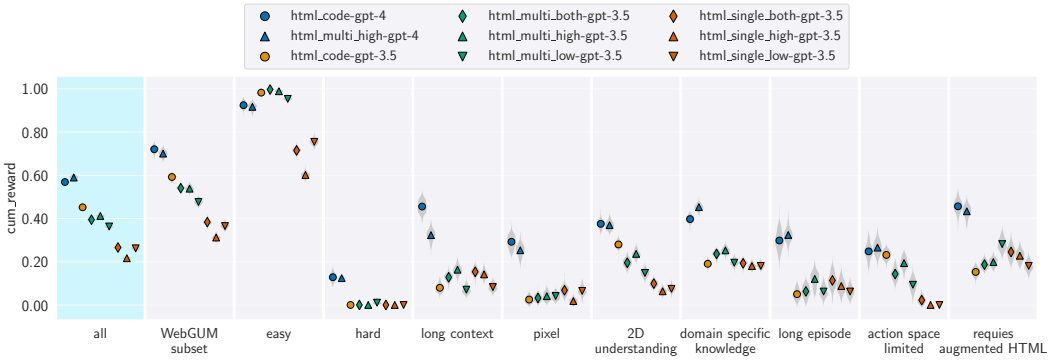


**Figure 2:** ***Action Space Analysis*** *– A per benchmark analysis of the action space. Green and red represent multi and single actions respectively. ▽, △, ◇, and ○ represent low-level, high-level, both and code actions respectively. The shaded region depicts statistical uncertainties from stratified bootstrap. Notably, the multi-action format exhibits consistent superior performance compared to the single-action format.*

Although our agents excel in several respects, they fall short of the performance set in prior work when applied to the WebGUM subset. This can be attributed to the zero-shot nature of our approach. Unlike previous research, which often involved specialized fine-tuning on expert demonstrations [Furuta et al.,

2023, Shaw et al., 2023] or reinforcement learning strategies [Kim et al., 2023], our methodology focuses on building general-purpose agents for the web and is not directly comparable to previous work.

### 5.4 Impact of the LLM

We continue our empirical analysis by constraining the observation space to HTML, thereby isolating the effects of LLM selection. This focused approach enables a more nuanced understanding of how LLMs impact the performance of web agents for specific tasks.

In our experiments, we utilize two open-source LLMs: Code Llama-34b-instruct-hf [Rozière et al., 2023] and StarChat $\beta$ 15B [Tunstall et al., 2023]. Code Llama, a specialized version of Llama-2 [Touvron et al., 2023b], undergoes an initial pretraining phase on a broad spectrum of natural language data before being fine-tuned on code, extended contexts, instruction-following datasets, and conversational data. In contrast, StarChat $\beta$ originates from StarCoder [Li et al., 2023] and is pretrained on a comprehensive collection of coding datasets. It is subsequently fine-tuned on natural language data, followed by instruction-following and chat datasets.

The contrasting training regimens of these LLMs offer valuable insights into the role of general knowledge versus coding expertise in their performance. Specifically, it is interesting to compare Code Llama, which dedicates the majority of its training to general knowledge acquisition, to StarChat $\beta$, where the focus is primarily on coding skills. It is worth noting that Code Llama is more than twice as large as StarChat $\beta$, resulting in a computational cost that is correspondingly higher.

The results, shown in Figure 3, demonstrate that open-source LLMs are viable alternatives to their closed-source counterparts. Moreover, finetuning these models on web tasks is expected to improve the success rate significantly. This not only bodes well for the open-source community but also serves as an impetus for researchers to include web tasks in their LLM evaluation benchmarks.

Interestingly, Code Llama notably excels in tasks requiring augmented HTML, while StarChat $\beta$ surpasses both Code Llama and GPT-3.5 in tasks that demand pixel reasoning. This suggests that StarChat $\beta$ may have a more refined internal rendering mechanism. Investigations are underway to further understand the origins of these observed behaviors.

On the downside, Code Llama encountered challenges in reliably generating Selenium code. We ascribe this issue to limitations in our current parser and possibly to a lack of depth in Selenium-specific knowledge. Active efforts are being made to refine the parser and resolve this shortcoming.

Lastly, GPT-4 demonstrates a pronounced advantage in handling long-context data. Given its hypothetical larger model size, we hypothesize that its additional computational capacity aids in processing more complex data.

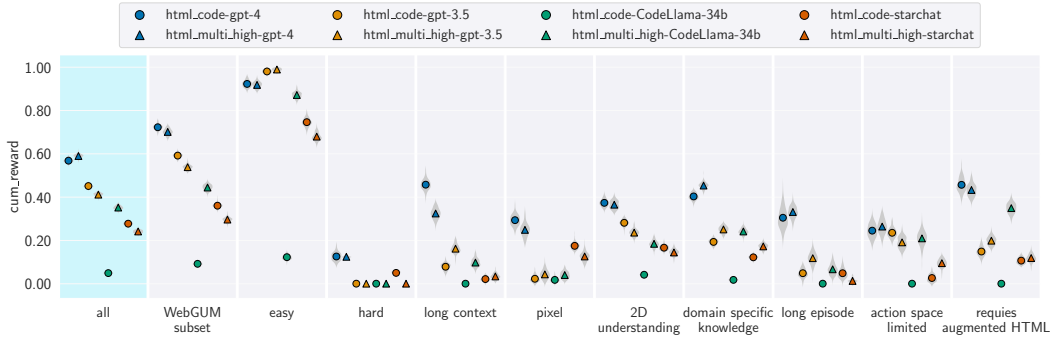


**Figure 3:** ***LLM Analysis*** *– A per benchmark analysis of the action space. Shapes represent the action space (code, vs multi_high). The shaded region depicts statistical uncertainties from stratified bootstrap. Encouragingly, the results suggest that open-source LLMs can serve as viable alternatives to their closed-source counterparts for generalist web agent tasks.*

### 5.5 Impact of the Observation Space

We conclude our empirical study by examining the observation space. Given that HTML files can grow exceedingly large on real websites, this becomes a significant bottleneck for LLMs with fixed

context lengths. Larger context sizes quadratically increase the computational overhead due to the quadratic complexity of attention mechanisms in transformers.

Thus, an LLM capable of operating on a more condensed version of the HTML, such as an accessibility tree, would be a noteworthy development.

In Figure 4, we present a comparison of the performance on all sub-benchmarks between HTML and accessibility trees across all four models.

Despite an observable decline in performance when utilizing accessibility trees, all agents maintain a respectable level of effectiveness on this more concise representation. Simple fine-tuning could potentially restore performance to HTML-levels, thereby presenting a computationally efficient alternative for agents.

In terms of computational speedup, we observe a reduction in the average prompt size from 2,501 tokens for the HTML agents to 1,529 tokens for those based on the accessibility tree. Owing to the quadratic complexity of the transformer's self-attention mechanism and the linear complexity of its other layers, we achieved a significant reduction in computational cost. Specifically, the computational cost for the self-attention and remaining layers decreased by factors of approximately 2.67 and 1.64, respectively.

One other advantage of using the accessibility tree is the lack of prior knowledge from the LLM on this data structure. This makes them more adaptable to new unconventional attributes and/or formatting. Notably, we have observed that LLMs often struggle to effectively leverage augmented information found in the HTML code, such as (`checked`, `hidden`, `value`...). We attribute this challenge to the model's tendency to become impervious to custom fields when dealing with supposedly well-established data structures like HTML.

As we can see in the results, methods relying on the accessibility tree constantly yield lower performance than their HTML counterpart. This drop is partly due to the fact that the MiniWoB web-tasks does not follow the ARIA best-practices and standards (re-purposing native HTML elements, no aria-attribute nor `<label>` tag listed), resulting in an incomplete representation. This observation underscores the importance of robustness in handling non-standard implementations, given the unpredictable nature of real-world websites. It highlights that the accessibility tree alone may not provide a comprehensive solution and suggests the need for an intermediate representation. This opens avenues for future work aimed at optimizing observation spaces for generalist web agents.

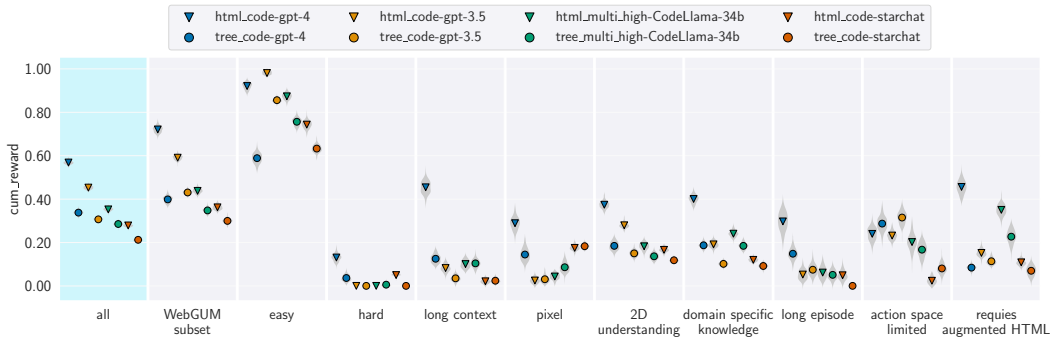


**Figure 4:** ***Observation Space Analysis*** *– While using accessibility trees results in a performance hit, they represent a more cost-efficient solution.*

## 6 Conclusion

Our research in employing Large Language Models (LLMs) as generalist agents for web applications has yielded numerous pivotal insights. We have observed that the choice of action space, specifically favoring code-based actions and employing multiple actions per timestep, enhances the performance of LLM-based agents, particularly in simpler tasks. Furthermore, the impact of LLM selection on agent capabilities is substantial, with open-source LLMs like Code Llama and StarChat offering viable alternatives to closed-source models like GPT-4, each excelling in specific task domains. Additionally, our exploration of observation space has highlighted the potential for LLMs to handle large sizes of HTML data while facilitating human-friendly interactions. These findings collectively underscore the

ongoing challenges and opportunities in the development of goal-driven AI agents for web applications and data processing, and emphasize the need for benchmarking platforms such as MiniWoB to advance research in this field. We restricted our study to the MiniWoB environment and laid down important axis that should be taken into account when designing generalist web agents. Indeed, the taxonomy of sub-benchmarks revealed a more detailed view on the relationship between tasks and cumulative rewards. Future work would extend our systematic analysis to larger-scale environments potentially spanning a wider range of challenges that we identified in section 3.

## References


Thomas Carta, Clément Romac, Thomas Wolf, Sylvain Lamprier, Olivier Sigaud, and Pierre-Yves Oudeyer. Grounding large language models in interactive environments with online reinforcement learning. *ArXiv*, abs/2302.02662, 2023.

Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. Guiding pretraining in reinforcement learning with large language models, 2023.

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

Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. A real-world webagent with planning, long context understanding, and program synthesis. *arXiv*, abs/2307.12856, 2023a.

Izzeddin Gur, Ofir Nachum, Yingjie Miao, Mustafa Safdari, Austin Huang, Aakanksha Chowdhery, Sharan Narang, Noah Fiedel, and Aleksandra Faust. Understanding HTML with large language models. *arXiv*, abs/2210.03945, 2023b.

Peter C Humphreys, David Raposo, Tobias Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Adam Santoro, and Timothy Lillicrap. A data-driven approach for learning to control computers. In *International Conference on Machine Learning (ICML)*, 2022.

Taichi Iki and Akiko Aizawa. Do BERTs learn to use browser user interface? exploring multi-step tasks with unified vision-and-language BERTs, 2022.

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

Michael Laskin, Luyu Wang, Junhyuk Oh, Emilio Parisotto, Stephen Spencer, Richie Steigerwald, DJ Strouse, Steven Stenberg Hansen, Angelos Filos, Ethan Brooks, maxime gazeau, Himanshu Sahni, Satinder Singh, and Volodymyr Mnih. In-context reinforcement learning with algorithm distillation. In *The Eleventh International Conference on Learning Representations*, 2023.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

Shuang Li, Xavier Puig, Chris Paxton, Yilun Du, Clinton Wang, Linxi Fan, Tao Chen, De-An Huang, Ekin Akyürek, Anima Anandkumar, Jacob Andreas, Igor Mordatch, Antonio Torralba, and Yuke Zhu. Pre-trained language models for interactive decision-making. In *Advances in Neural Information Processing Systems*, 2022.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *arXiv preprint arXiv:2209.07753*, 2022.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference of Learning Represeantation (ICLR)*, 2018.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. *arXiv*, abs/2308.03688, 2023a.

Zhiwei Liu, Weiran Yao, Jianguo Zhang, Le Xue, Shelby Heinecke, Rithesh Murthy, Yihao Feng, Zeyuan Chen, Juan Carlos Niebles, Devansh Arpit, Ran Xu, Phil Mui, Huan Wang, Caiming Xiong, and Silvio Savarese. Bolaa: Benchmarking and orchestrating LLM-augmented autonomous agents. *arXiv*, abs/2308.05960, 2023b.

Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, pages 2661–2670, 2017.

OpenAI. GPT-4 technical report. *ArXiv*, abs/2303.08774, 2023.

Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: Outperforming curated corpora with web data, and web data only, 2023.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023.

Peter Shaw, Mandar Joshi, James Cohan, Jonathan Berant, Panupong Pasupat, Hexiang Hu, Urvashi Khandelwal, Kenton Lee, and Kristina Toutanova. From pixels to UI actions: Learning to follow instructions via graphical user interfaces. *arXiv*, abs/2306.00245, 2023.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In *International Conference on Machine Learning (ICML)*, 2017.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023a.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023b.

Lewis Tunstall, Nathan Lambert, Nazneen Rajani, Edward Beeching, Teven Le Scao, Leandro von Werra, Sheon Han, Philipp Schmid, and Alexander Rush. Creating a coding assistant with starcoder. *Hugging Face Blog*, 2023. https://huggingface.co/blog/starchat.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi (Jim) Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *ArXiv*, abs/2305.16291, 2023.

World Wide Web Consortium. Document Object Model (DOM) Level 3 Core Specification, 2004. URL https://www.w3.org/TR/DOM-Level-3-Core/. Accessed: 2023-09-25.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *arXiv*, abs/2210.03629, 2023.

Longtao Zheng, Rundong Wang, and Bo An. Synapse: Leveraging few-shot exemplars for human-level computer control. *arXiv*, abs/2306.07863, 2023.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. WebArena: A realistic web environment for building autonomous agents. *arXiv*, abs/2307.13854, 2023.

# A Supplementary Material

## A.1 Cherry-picked Examples

In this section we show a set of cherry-picked examples representative of the bottlenecks we identifed in our study when it comes to designing web agents. We specifically put the focus on examples showcasing the advantage of having code as action space in 6 and 8, limitation of agents in terms of domain-specific understanding of web elements 8 and 9, and where a visual understanding would help 9 or is necessary to solve the task at hand 10. On an another note, 7 shows the impact of the browser when performing an action. When clicking in the middle of an `input` field, Safari and Chrome will open up a custom calendar window, allowing the user to select the appropriate date. While being sometimes overlooked in the literature, we noted in our experiment that this browser overlay can sometimes highly confuse our vision-impaired agent.

## A.2 Action and observation space description

We describe in this section the details of both the action space and the observation space.

**Action Space** Our action space spans a pre-defined range of actions that we list below :

- **Click** : represents the action of clicking on a web element, triggering its default behavior (e.g., opening a link).
- **Type**: involves inputting text into a selected web element, such as text fields or forms.
- **Clear**: clears any existing content from a selected input field.
- **Hover**: simulates hovering the mouse cursor over a web element without triggering a click event.
- **Key Press**: mimics pressing a specific keyboard key or combination of keys.
- **Scroll**: allows scrolling within a web page, either vertically or horizontally.
- **Mouse up** : releases the mouse primary button.
- **Mouse down** : hold the mouse primary button.

As elaborated in Section4 we further distinguish between low-level actions and high-level actions set based on the method of element/location identification. This distinction applies to the following subset of action types : click, type, clear, mouse up/down and hover. In this subset, element/location identification can be done through :

- A high-level bid attribute which uniquely identifies elements of the DOM.
- Low-level x,y coordinates which can either correspond to the center coordinates of an element of the DOM or any location in the viewport.

Low-level identification of an arbitrary location in the viewport is particularly useful for tasks that require spatial interactions. A good representative of such tasks is the `bisect-angle` task where the agent needs to click in between the two html elements corresponding to the lines.

Moreover, our framework allows agents to leverage both high-level and low-level element/location identification methods within the same task. These distinctions are facilitated through the use of prompts provided to instruct the agent for a specific task, which we describe in the subsequent section. We describe in detail the different prompts in the following section.

**Observation Space**

For defining a text-based approach, we provide to the agent either the HTML source code of the webpage with additional information or a token-efficient data-structure called the accessibility tree.

- **HTML** : In this case, we provide the HTML source code of the webpage, augmented with additional HTML attributes. These attributes include:
  - `bid`: A unique identifier assigned to each element in the DOM tree.
  - `isinviewport`: A boolean attribute indicating whether the element is currently visible within the current window.

- `viewport_x` and `viewport_y`: These attributes represent the $(x, y)$ coordinates of the center of the element within the current window.
- `value`: This attribute stores the actual value for any `<input>` element present in the DOM tree.
- `checked`: A boolean attribute applicable to `<input type="checkbox">` elements in the DOM tree.

These additional attributes enhance the HTML source code, providing the agent with valuable context and data for its operations.

- **Accesibility tree** : In this case, we provide the accessibility tree. [14] The computed accessibility tree for a simple page is presented Fig. 11.

### A.3 Prompts details

We describe here the exact content of the different prompts used for the experiments. We distinguish between different parts constituting the prompts :

- Action space description
- Observation space description
- Additional information and general guidelines
- Output formatting description

As explained in previous sections, the action space description can refer to high-level, low-level, both or code action space. We give below the description of the high-level, low-level and code action space.

**High level element identification** :

```
hover : move the mouse over an html element identifed with its bid attribute.
Examples:
{"action_type": "hover", "bid": 857}
click : mouse click on an html element identifed with its bid attribute..
Examples:
{"action_type": "click", "bid": 24}
mouse_down: move the mouse to an html element identifed with its bid attribute,
then clicks and hold the mouse primary button.
Examples:
{"action_type": "mouse_down", "bid": 98}
mouse_up: move the mouse to an html element identifed with its bid attribute,
then release the mouse primary button.
Examples:
{"action_type": "mouse_up", "bid": 98}
type: if specified, clicks an html element identifed with its bid attribute,
then clears any previous text in the focused element (if applicable),
and types 'text' via the keyboard.
Examples:
{"action_type": "type", "bid": 132, "text": "Hello"}
{"action_type": "type", "text": "Paul"}
clear: if specified, clicks an html element identifed with its bid attribute,
then clears any previous text in the focused element (if applicable).
Examples:
{"action_type": "clear", "bid": 132}
scroll: scroll the page in the specified direction by the specified amount.
Examples:
{"action_type" : "scroll", "direction" : "down", "amount":5}
press_key: press the specified key combination.
Examples:
{"action_type" : "press_key", "key_comb" : "C-a"}
```

[14] Please visit https://developer.mozilla.org/en-US/docs/Glossary/Accessibility_tree for more details.

**Low-level element/location identification** :

```
hover : move the mouse to some specified coordinates x and y in the viewport.
Examples:
{"action_type": "hover", "x": 87, "y": 158.9}
click : mouse click on some specified coordinates x and y in the viewport.
Examples:
{"action_type": "click", "x": 66.1, "y": 32.5}
mouse_down: move the mouse to to some specified coordinates x and y in the viewport,
then clicks and hold the mouse primary button.
Examples:
{"action_type": "mouse_down", "x": 458.3, "y": 775.2}
mouse_up: move the to some specified coordinates x and y in the viewport,
then release the mouse primary button.
Examples:
{"action_type": "mouse_down", "x": 458.3, "y": 775.2}
type: if specified, clicks an html element identifed with its bid attribute,
then clears any previous text in the focused element (if applicable),
and types `text` via the keyboard.
Examples:
{"action_type": "type", "x": 597.5, "y": 3.5, "text": "Hi!"}
{"action_type": "type", "text": "Paul"}
clear: if specified, mouse click on some specified coordinates x and y in the viewport,
then clears any previous text in the focused element (if applicable).
Examples:
{"action_type": "clear", "x": 59.5, "y": 8.5,}
scroll: scroll the page in the specified direction by the specified amount.
Examples:
{"action_type" : "scroll", "direction" : "down", "amount":5}
press_key: press the specified key combination.
Examples:
{"action_type" : "press_key", "key_comb" : "C-a"}
```

**Code Action space description** :

```
You have access to a selenium webdriver local variable named driver.
Use the latest version of the selenium API with the generic find_element(By.) locator.
You can use the bid to uniquely identify and select an element like

```driver.find_element(By.CSS_SELECTOR, '[bid=\"3\"]')```
to select the element with bid 3.

The following selenium libraries are already imported and you can use them :
from selenium.webdriver.common.by import By
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
You must not import additional libraries
```

We then list the different output formatting description and distinguish between single JSON-like action, multiple JSON-like action and selenium code formatted output per LLM-call.

**Single JSON-like Action formatting** :

```
You must output each action in a JSON format with the action_type key
and the rest of the arguments to give to the action.
You must output the next single action needed to solve the task at hand.
Output Example :
{"action_type": "click", "bid": "1234"}
```
```

**Multplie JSON-like Action formatting** :

```
You must output each action in a JSON format with the action_type key
and the rest of the arguments to give to the action.
You must output a list that contains the next sequence of actions
needed to solve the task.
Output Example:
[{"action_type": "click", "bid": "1234"},
{ "action_type": "type","bid": "123", "text": "Hello World"}]
```

**Selenium code formatting** :

```
You must use valid selenium code only in your answer. Example Output :
'''
driver.find_element(By.CSS_SELECTOR, '[bid="3"]')\nelement.click()
'''
```

**General Guidelines** We additionally consider prompting the agents with general guidelines to take into consideration when interacting with web elements.

```
Here are some general guidelines:
1. Understand the nature of web elements, such as buttons,
input fields, date pickers,
and dropdowns. Recognize which elements are clickable, editable, or selectable.
2. Interact with web elements in a way that simulates human interaction.
3. Be aware that some web elements might not be immediately interactable.
You may need to wait for them to become available.
4. Check whether your previous actions had the intended effect.
If not, you must try something different and decompose step by step your solution.
```

**Additional Information** We also include additional information about the nature of the agent observation space such as the accessibility description :

```
The web page is presented in a customized format that is generated from the
accessibility tree.
In this format, each element on the page is described with a set of attributes.
These attributes include:
value of Input Element: For input elements such as text fields or checkboxes,
there is an attribute that indicates the current value entered or selected.
disabled Status: Some elements may have an attribute indicating whether they are
disabled or not.
This helps identify if certain actions can be performed on those elements.
focused Element: An attribute may indicate if an element currently has focus.
This is crucial for understanding which element is currently active or selected.
(x,y) coordinates: The position of each element on the page is provided in terms of
its coordinates.
This information is valuable for understanding the element's position
in relation to others.
[bid] bid css attribute indicated between brackets before each element.
```

Note that we also augment the html with existing dynamic attributes such as the value of an input field but do not specifically mention it to the agent since we do not rename the attribute. The notion of bid as unique css identifier in the html is introduced through the action space description.

### A.4 Ethical Considerations and Risk Assessment

While LLM-based web agents have the potential to perform a wide range of tasks autonomously, there are significant ethical considerations and risks associated with their deployment. We list a few of them in this section. One notable concern is the possibility of these agents to inadvertently disseminate false or biased information. Since LLMs rely on the vast amount of data they have been trained on, there is a risk that they may generate content that contains inaccuracies, misinformation, or perpetuates existing

biases present in the training data. This could lead to the unintentional spread of false information, contributing to the proliferation of misinformation on the web. The use of LLM-based agents can also raise intellectual property issues, as they can generate content that may infringe on copyright and intellectual property rights without proper authorization. Additionally, web agents that perform tasks autonomously can consume significant computing and network resources, potentially leading to unintentional denial-of-service (DoS) attacks on web servers. Finally, an issue directly linked to designing agents that can act as UI assistants is their exposure to potentially sensitive information about the user resulting in risks such as data theft or their account being compromised. It is therefore crucial to implement safeguards and validation mechanisms to mitigate these risks and ensure that web agents prioritize accuracy and objectivity in the information they generate and disseminate, while also respecting intellectual property rights, data privacy and resource consumption constraints.

Some additional ethical considerations also play a pivotal role. Among them transparency and accountability are essential, ensuring that there is an alignment between the user intent and the actions executed by the agent. Users must therefore be informed about the agent's capabilities and limitations to make informed decisions regarding its usage. It is also crucial to ensure that the agent's actions adhere to local and international laws, safeguarding against any unintended legal violations. Additionally, the environmental impact of such agents should not be overlooked, as they can consume significant computational resources, contributing to increased power usage and a larger carbon footprint. Striking a balance between efficiency and sustainability in the design and operation of LLM-based web agents is an ethical imperative, aligning with broader environmental goals.

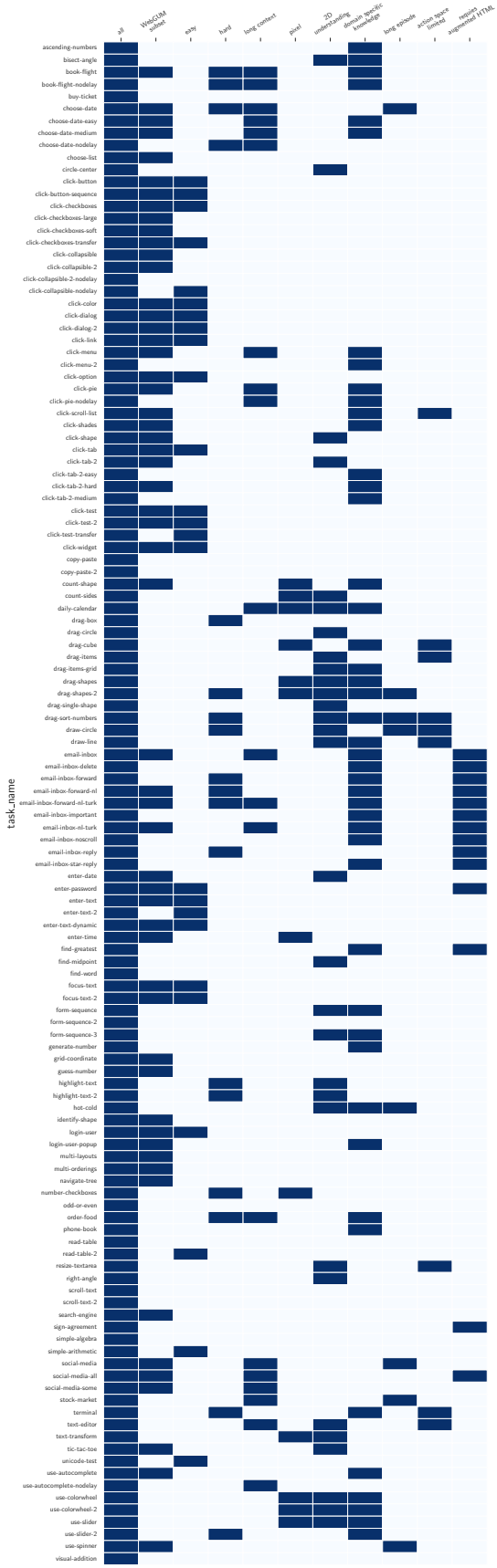


**Figure 5:** *Complete list of tasks for MiniWob, showing the subset of tasks introduced in the section 5.2, Sub-benchmark Analysis.*

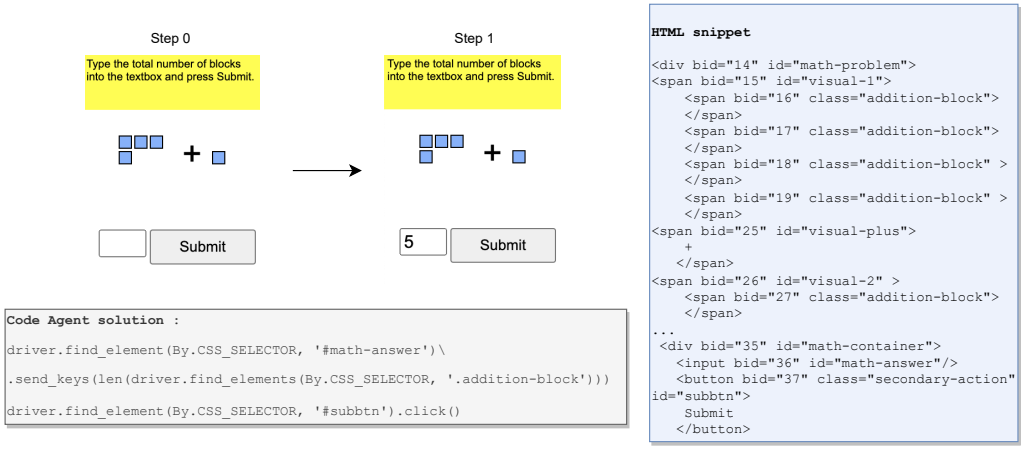


**Figure 6:** *Code Efficiency. This example shows the benefit of having a flexible code action space. The agent can leverage code efficiency to translate the reasoning part as code execution. It automatically counts the number of squares and put the result in the appropriate answer box.*

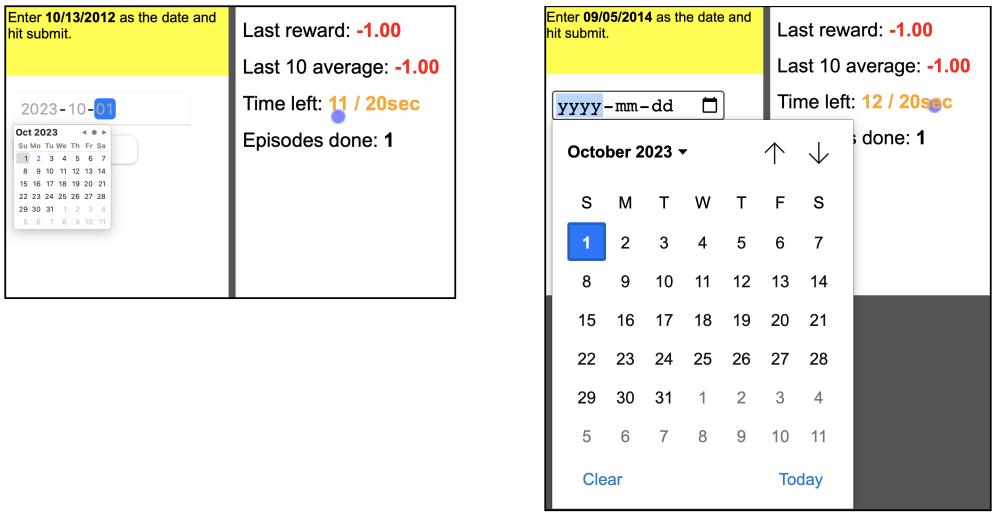


**Figure 7:** *Different date-picker widget depending on the browser used. On the left: Safari. On the right: Chrome. This pop-up is actually generated by the browser itself, and won't be found in the Document Object Model (DOM) of the webpage.*

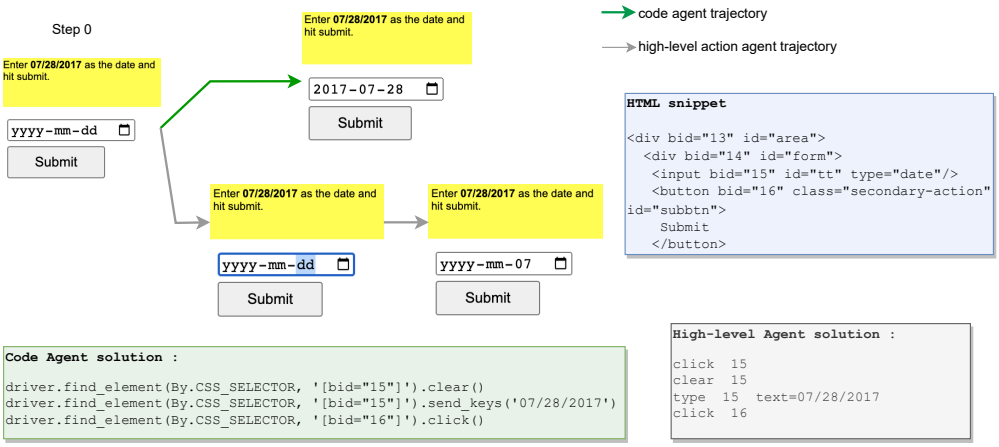


**Figure 8:** *Domain specific knowledge. This example shows a case where the agent needs some domain specific knowledge about the input type date as a web element. Specifically selecting the input element with its bid would place the cursor at the center of the text box (eg. and only select the day* dd *value). In order to solve the task, the action needs to know how to interact with this element by either having the domain specific knowledge on the date input type behavior or relying on visual cues to understand it.*

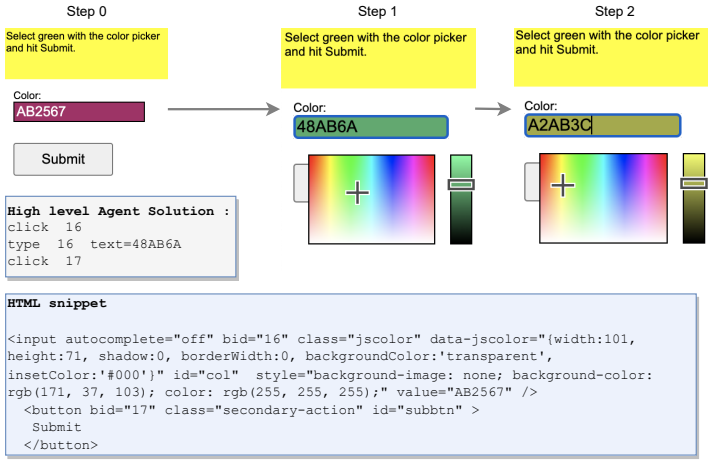


**Figure 9:** *Domain specific knowledge and visual limitation. This example shows a domain-specific behavior of the color picker web element. The agent understands the task at hand and type the right color in the input but does not consider the fact that typing a value in the input element will trigger a widget that is interactable with a cursor and hides the submit button. On the other hand, a visual cue would help the agent understand this behavior that is not necessarily inferable form the html state only, unless the llm has been trained with this family of input date web element.*

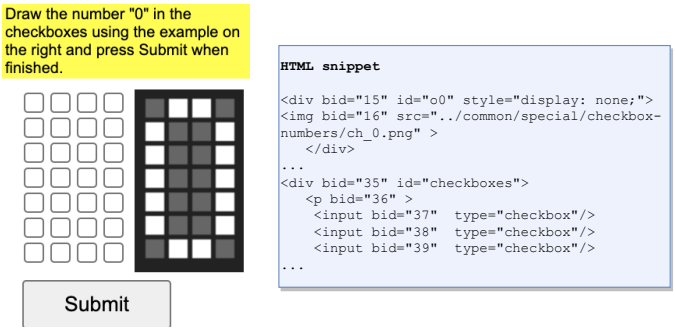


**Figure 10:** *Visual Limitation. This is example where the task is not solvable without the image target encapsulated in an <img> tag.*

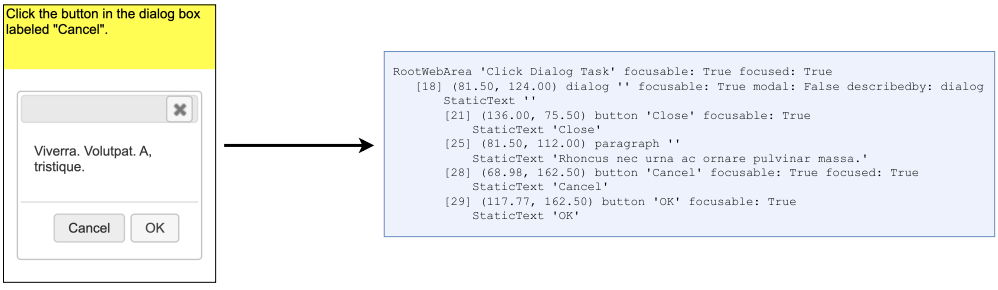


**Figure 11:** *Example of an accessibility tree obtained on the MiniWoB task click-dialog-2*