

# WORKARENA: HOW CAPABLE ARE WEB AGENTS AT SOLVING COMMON KNOWLEDGE WORK TASKS?

Alexandre Drouin<sup>12\*</sup> Maxime Gasse<sup>123\*</sup> Massimo Caccia<sup>1</sup> Issam H. Laradji<sup>1</sup>  
Manuel Del Verme<sup>24</sup> Tom Marty<sup>125</sup> Léo Boisvert<sup>13</sup> Megh Thakkar<sup>125</sup> Quentin Cappart<sup>23</sup>  
David Vazquez<sup>1</sup> Nicolas Chapados<sup>123</sup> Alexandre Lacoste<sup>1</sup>  
<sup>1</sup>ServiceNow, <sup>2</sup>Mila, <sup>3</sup>Polytechnique Montréal, <sup>4</sup>McGill University, <sup>5</sup>Université de Montréal

## ABSTRACT

We study the use of large language model-based agents for interacting with software via web browsers. Unlike prior work, we focus on measuring the agents’ ability to perform tasks that span the typical daily work of knowledge workers utilizing enterprise software systems. To this end, we propose WorkArena, a remote-hosted benchmark of 29 tasks based on the widely-used ServiceNow platform. We also introduce BrowserGym, an environment for the design and evaluation of such agents, offering a rich set of actions as well as multimodal observations. Our empirical evaluation reveals that while current agents show promise on WorkArena, there remains a considerable gap towards achieving full task automation. Notably, our analysis uncovers a significant performance disparity between open and closed-source LLMs, highlighting a critical area for future exploration and development in the field.

## 1 INTRODUCTION

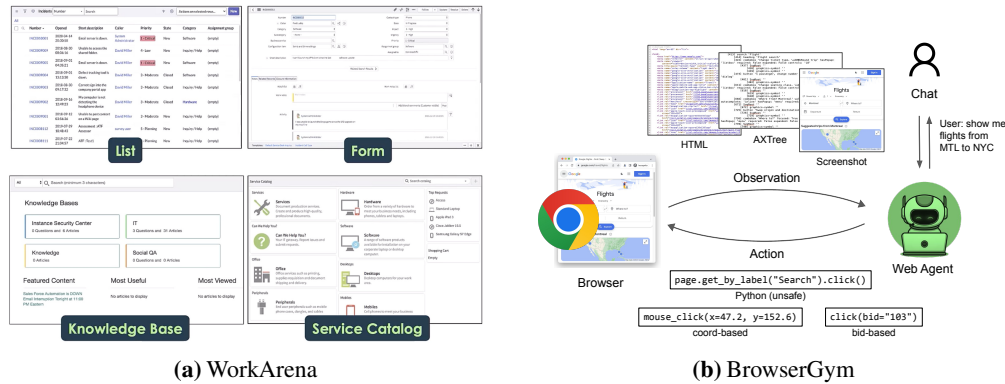
Graphical User Interfaces (UIs) are the predominant medium through which people interact with software, serving as a crucial gateway to the digital world. While they have evolved to become more intuitive, featuring generic and universal components like forms, lists, and buttons, UIs can still make complex or repetitive tasks burdensome for users. While being more and more intuitive, those complex UIs also became inadvertently more and more discriminative for visually-impaired users. An ideal user experience would involve automated assistants that can streamline these tasks ensuring accessibility for everyone. While Application Programming Interfaces (APIs) have facilitated programmatic interactions with software, the resulting automated assistants often lack transparency, are difficult for users to inspect, and are not universally available. In contrast, assistants that directly manipulate UIs (UI assistants) offer greater transparency and are more amenable to human oversight. Most notably, because the user can give and take back control over the UI at any point, UI assistants can provide varying levels of automation ranging from partial assistance (such as finding a menu or filling a form) to complete task execution (like placing an order), akin to the six levels of automation in autonomous driving (SAE, 2021).

Recent advancements in the fields of large language and vision models have seen the rapid development of UI assistants, particularly *web agents* acting through a browser (Furuta et al., 2023; Kim et al., 2023; Gur et al., 2023b). The range of web tasks explored in the literature varies from simple UI commands such as selecting specific menu elements on toy web pages (Liu et al., 2018; Shi et al., 2017a), to more complex requests such as “Checkout merge requests assigned to me”, on real-world websites like Reddit and GitLab (Zhou et al., 2023). Yet, one area in which web agents can be particularly impactful and remain unexplored is enterprise software. In the workplace, where repetitive tasks are common, enterprise software often prioritizes functionality over user experience, leading to inefficiencies and long learning curves for workers. Our work addresses this gap, and investigates the potential of web agents in enterprise settings to improve accessibility, user experience, and worker productivity.

To this end, we introduce *WorkArena*, a benchmark developed on the widely-used ServiceNow platform (ServiceNow, 2023). ServiceNow is a comprehensive cloud-based platform that offers solutions for automating and managing digital workflows across various enterprise functions, including IT service management, human resources, customer service, and security operations. In 2023

---

\*Equal contribution.



**Figure 1:** Overview of contributions: (a) **WorkArena** is a benchmark of 29 web tasks and 18,050 unique instances that cover common ways of interacting with the ServiceNow Platform, a widely-used enterprise software platform. (b) **BrowserGym** is a Python environment for designing and evaluating web agents, which includes a rich set of actions and multimodal observations (shown here the HTML contents of the page, its accessibility tree, and the raw pixels after browser rendering).

their customer base counted over 7,000 companies worldwide, including 85% of the Fortune 500 companies (Mastantuono, 2023). Within these firms alone, the ServiceNow platform potentially impacts over 12 million individuals, not including broader public interactions, such as the 500,000 daily users of Disney+’s customer help center (Maas, 2020). ServiceNow’s extensive reach makes it an ideal real-world environment for evaluating the potential impact of UI assistants in the workplace.

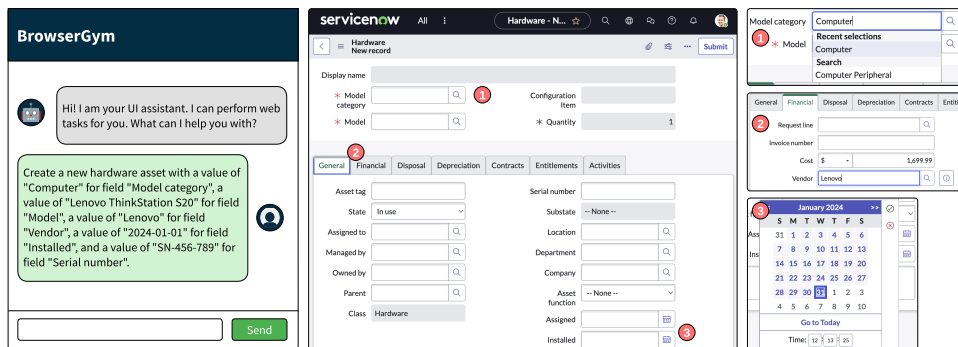
Our contributions are as follows:

- **WorkArena:** A realistic benchmark of enterprise-related tasks for web agents comprising 18,050 unique task instances (§ 3, Fig. 1a);
- **BrowserGym:** A new framework for the development and evaluation of web agents, compatible with previous benchmarks like WebArena (Zhou et al., 2023), MiniWoB (Liu et al., 2018; Shi et al., 2017a) and WebShop (Yao et al., 2022), that offers a richer set of multimodal observations (e.g., screenshot, accessibility tree, screen coordinates), a broader set of actions (e.g., Python code and high-level primitives), and supports chat-based interactions (§ 4, Fig. 1b); Surprisingly, these features contribute to bringing our GPT-4 agent at the top of the leaderboard on WebArena, with a score of 25.4%, contrasting with the score of the original paper, 14.4%.
- **Empirical study:** We report a collection of experiments to assess the ability of state-of-the-art large language model (LLM)-based agents to solve WorkArena, as well as an analysis of the impact of the different BrowserGym features on WorkArena and MiniWoB (§ 5).

## 2 RELATED WORKS

**Benchmarks for web agents:** Early benchmarks for web agents were based on synthetic web environments where agents were tasked with performing low-level keyboard and mouse actions (Shi et al., 2017b). Notable examples are MiniWoB (Shi et al., 2017a; Liu et al., 2018), which offer a collection of 125 toy web tasks ranging from clicking a specific button to using a basic text editor, and WebShop (Yao et al., 2022), a simulated e-commerce website with shopping tasks that require searching and browsing a catalog of items. More recently, Zhou et al. (2023) introduced WebArena, a collection of 190 tasks based on realistic websites that emulate real-world domains such as e-commerce, social forums, collaborative software development, and content management. WebArena is a notoriously challenging benchmark, with a success rate of 14% for a state-of-the-art web agent based on GPT-4, and 78% for human agents. Deng et al. (2023) proposed Min2Web, a large-scale dataset of 2,000 web interactions from 137 websites curated by human annotators. Similarly, Lù et al. (2024) propose WebLINX, a curated dataset of web interactions composed of 2337 expert demonstrations from 155 different real-world websites. In WebLINX, each task is composed of a turn-based chat dialogue averaging 43 interactions per task. Last, He et al. (2024) propose 300 information-retrieval tasks, from 15 real-world consumer websites (e.g., Amazon, Coursera, Booking), which are used to evaluate WebVoyager, a vision-based web agent.

Our proposed benchmark, WorkArena, is designed to complement existing work by specifically focusing on real-world enterprise software applications. It includes a wide range of tasks that



**Figure 2:** Example form task – The goal is given to the agent in natural language via the chat interface. As can be seen, the goal is designed to be very explicit, leaving no ambiguity on the task to perform. As for the UI, it is complex, composed of many fields, some of which are dynamic, such as auto-completion-based text boxes ①, some are hidden behind tabs ②, and others require complex interactions, such as date pickers ③. Other such examples are available in § A.2.

collectively encompass several end-to-end workflows typically performed by knowledge workers. Additionally, it poses a series of technical challenges, such as pages with very large document object models (DOMs), non-standard HTML, and complex UI elements, which we outline in § 3.2. This benchmark integrates into BrowserGym, a new environment that we propose for the evaluation of web agents, which aggregates all features proposed in previous work, such as multimodal observations and code-based actions while being the first to support chat-based agent-user interactions (§ 4.1).

**LLM-based Agents:** The scope of our experimental contributions is limited to web agents that rely on language models for reasoning. Recent studies include the seminal work of Nakano et al. (2021) that introduces WebGPT, an agent capable of browsing the web and answering questions via information retrieval. Other works have also explored web agents that receive HTML as input and produce a series of high-level actions such as *click*, *type*, *select* (Deng et al., 2023; Liu et al., 2023a;b; Yao et al., 2023).

Other works have shown that using just text content as input is limited information, and, therefore, have considered multimodal observations, which includes both visual (screenshots of a page) and text content in order to perform a task (Humphreys et al., 2022; He et al., 2024). Instead of directly interacting with a website, recent works have proposed methods that can act on websites using Python-generated code from task-specific instructions (Gur et al., 2023a;b). Our proposed environment, BrowserGym is flexible in that it supports all observations and actions spaces utilized in prior research.

### 3 WORKARENA – AN ENTERPRISE BENCHMARK

WorkArena consists of a suite of 29 tasks and 18,050 unique instances that cover core interactions with the ServiceNow platform, such as navigating lists, filling forms, searching knowledge bases, utilizing service catalogs, and navigating via menus (see Fig. 1a). Collectively, these tasks are representative of a wide array of common operations that employees, like IT, administrative, and white-collar staff, perform on a daily basis.

As a guiding example, consider an IT support agent tasked with onboarding new hires. Each day, this agent logs into the ServiceNow platform. They use the **menu to navigate** to a list of requests to be fulfilled. They plan their work by **filtering** the list to extract all requests assigned to them and **sorting** the list in order of priority. They then process requests by **filling out forms** to create new user profiles, and use the **service catalog** to order laptops for them. As we will see, all of the interactions listed above are included in WorkArena, and this is only one of the many user trajectories that the benchmark covers.

#### 3.1 WORKARENA TASKS

In WorkArena, each task is coupled with a natural language goal that provides unambiguous instructions to the agent (examples are illustrated in Fig. 3). The benchmark includes validation functions that offer real-time feedback to agents, identifying errors ranging from minor (e.g., unfilled mandatory fields) to critical (such as pushing invalid data to the database). A unique feature of WorkArena in contrast with prior benchmarks is the inclusion of a *cheating function* for each task,

implemented using Playwright browser automation (Microsoft, 2023). This function, designed to complete the tasks successfully, serves three purposes: (i) it ensures the feasibility of each task, (ii) it serves as the ground truth for agents that have learning capabilities, and (iii) it helps maintain the benchmark’s longevity by making it easier to identify and adjust tasks impacted by future updates to the ServiceNow platform. The WorkArena benchmark runs on ServiceNow developer instances, which are freely available (with limited capacity) from <https://developer.servicenow.com>, and initialized to a known state with demonstration data, ensuring consistency and reproducibility of benchmark runs.

**Lists:** We consider 12 list-based tasks, which can be grouped into two categories: filtering and sorting. The former consists of using the UI to construct a complex filter with 1 to 5 conditions. The latter consists of using the UI to sort the list based on up to 3 columns. In both cases, the interaction with the UI is non-trivial and requires opening a hidden menu, adding the right number of conditions, and filling them out accordingly. There are 6 tasks of each type, each corresponding to a different data table (e.g., users, incidents). In both cases, client-side validation is used to verify that the resulting list satisfies the expected conditions. Together, these tasks yield 6,900 instances.

**Forms:** We consider 5 form-based tasks, which each consist in creating a new entry in a given data table. These tasks vary in complexity based on the number of fields that must be filled (from 1 to 26) and on the intricate properties of each form’s UI. For example, some forms require navigating through a set of tabs to expose hidden fields. Others use dynamic auto-completions fields, which require careful handling (see Fig. 2 for an example). In all cases, validation proceeds by querying the database to retrieve entries created by the agent and verifying that their values are as expected. Together, these tasks yield 5,000 instances.

**Knowledge bases:** The benchmark includes an information retrieval task that consists of searching the platform’s knowledge base to answer a question. Concretely, this requires searching with appropriate keywords and browsing the resulting articles to find specific information. These tasks are constructed by starting from a list of facts, generating articles containing each fact with GPT-4 (OpenAI, 2023), and generating a series of questions that unambiguously ask for this fact. Then, validation proceeds by verifying if the answer returned by the agent is within a set of acceptable answers. For example, if the question is “What is the level of customer satisfaction?” and the answer is “8.5/10”, alternative answers such as “85%” or “8.5 out of 10” would be accepted. In total, this task yields 1,000 instances. Details on article generation and validation are given in § A.3.

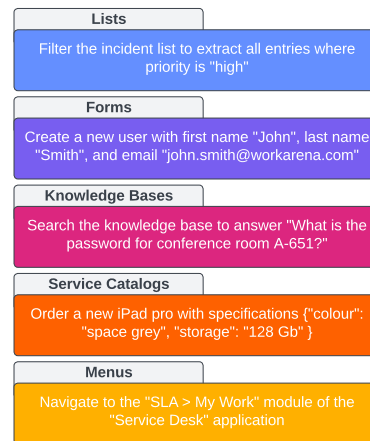
**Service catalogs:** The benchmark includes 9 tasks that require navigating a catalog of products and ordering items with given specifications. Such tasks vary in complexity based on the number of item configuration options. In all such tasks, validation is done by querying the database to verify that the order request created by the agent includes the right items in the right amounts, with the expected specifications. Together, these tasks yield 3,550 instances.

**Menus:** We consider 2 menu-based tasks: i) navigating via the “All” menu and ii) impersonating users. The first consists of using the platform’s main menu to navigate to a given application. In this case, validation simply verifies that the agent has arrived at the expected location. The second consists of *impersonating* a user, a task commonly performed by IT support agents, where the agent logs into the platform as a given user to diagnose an issue. In this case, validation verifies that the expected user is logged in. Together, these tasks yield 1,600 instances.

A detailed list of all the tasks implemented in WorkArena is available in § A.1 Tab. 7.

### 3.2 CHALLENGES: THE WORLD *Wild* WEB OF WORK

The ServiceNow platform poses a specific set of challenges for UI assistants, which we believe make WorkArena a complementary and meaningful benchmark for the community.



**Figure 3:** Example goal for each kind of WorkArena task.

**Non-standard, dynamic UIs:** First, the web pages are heavily dynamic, and exhibit complex UI elements and ways of interacting with them. For example, the form to create an incident contains specific rules that can make some fields required or hidden, depending on the value of other fields (e.g., setting an incident’s status to “Resolved” requires filling its “Resolution notes”). Or, on some pages, the right-click can be overloaded to display a dynamic menu in certain areas. While these UI behaviors are not necessarily standard or best practices in web development, they are fairly common and representative of real-world enterprise software, which is not always designed with user accessibility in mind.

**Non-standard, exotic DOMs:** Second, the ServiceNow platform relies on a complex combination of web technologies to implement its web pages, with nested iFrames, shadow DOMs, and proprietary Javascript APIs and HTML tags that do not necessarily adhere to existing web standards.<sup>1</sup> This specificity would require strong out-of-distribution generalization for a UI assistant to successfully solve a task.

**Large DOM trees:** Third, the HTML Document Object Model (DOM) of rendered web pages in the ServiceNow platform can be prohibitively large even for state-of-the-art language models, with a flat text size that ranges between 40k and 500k tokens, even after a basic cleaning (removing scripts, styles and empty elements). Thus, even a conceptually simple task, such as finding the next element to click on the current page requires long context understanding, which is an active area of research in language models.

## 4 BROWSERGYM

Along with the WorkArena benchmark, we introduce BrowserGym, a generic browser environment that facilitates the design of new benchmarks, and provides a solid platform for the evaluation of multi-modal web agents. BrowserGym (Fig. 1b) is implemented as an OpenAI Gym environment (Brockman et al., 2016) and follows a Partially-Observable Markov Decision Process (POMDP) paradigm.<sup>2</sup> It relies on Chromium and uses the Chrome DevTools Protocol (CDP) (Google, 2023) and the Playwright library (Microsoft, 2023) to interact with the web browser.

### 4.1 CAPABILITIES

BrowserGym implements the following capabilities.

**Chat-based user interaction:** one of the interaction modalities is a chat interface where the user and the web agent can exchange messages. In WorkArena, the goal of each task is provided as the initial user message, to which the agent can reply at any time. This allows for information retrieval tasks where a specific answer is expected from the agent, but also sequential tasks where user instructions change over time and are delivered sequentially in a way that mimics real-world use cases.

**Augmented DOM attributes:** BrowserGym marks every element on the web pages with a unique identifier `bid`, its center screen coordinates  $(x, y)$ , its bounding box  $(left, top, right, bottom)$ , and a visibility flag `(visible)`. These attributes provide a crude summary of the visual rendering of the UI and allow unambiguous interaction with individual elements through their identifiers.

**Rich observation space:** at each time step, the observation space contains the content of the chat (list of messages), the currently open pages (list of URLs), the error message from the last action if any (stack trace), and a multi-modal view of the active web page: its DOM snapshot (structured object), its accessibility tree or AXTree (structured object) as originally proposed by Zhou et al. (2023), and a viewport screenshot (image). Both DOM and AXTree are enriched with augmented attributes (`bid`, `coords`, `visible tag`) and are obtained through Chrome’s CDP. These structured objects can typically be rendered as text for processing by a language model and can be combined with the screenshot for a vision-language model.

**Rich action space:** the action space is customizable and includes Python code, which can be restricted to specific set of pre-defined high-level primitives, such as `bid`-based actions (`click(bid)`, `type(bid, text), ...`), and `coord`-based actions (`mouse_click(x, y)`, `keyboard_type(text), ...`).

<sup>1</sup><https://www.w3.org/standards/>

<sup>2</sup><https://github.com/ServiceNow/BrowserGym>

Alternatively, the action space be left unrestricted and allow the execution of arbitrary Python code, including the entire Playwright API and giving the web agent maximum flexibility in interacting with the browser. For the complete list of high-level primitives available in Browsergym, refer to § B, Tab. 8.

**Multi-page navigation:** BrowserGym natively supports web tasks that require multiple open pages (tabs, popups) and is also robust to web pages that employ nested encapsulation techniques such as iFrames and shadow DOMs. This robustness is essential to handle the heterogeneity of real-world websites and is missing in existing web environments.

## 4.2 AN IDEAL EXPERIMENTAL FRAMEWORK

**Flexible agent design:** BrowserGym offers an extensive list of features but does not impose any restriction on how web agents should be implemented. The agent is responsible for using the provided observations or not (DOM, AXTree, screenshot, error message), deciding how to handle the history (past observations and actions), or deciding which action space it should be using (python, bid, coord, coord+bid). As such, with BrowserGym, researchers can easily experiment with new ideas and evaluate and compare a wide variety of web agents on the same set of tasks, such as text-only agents, vision-augmented agents, memory-augmented agents, and so on.

**Minimal task design:** BrowserGym reduces the burden of creating new benchmarks to a minimum. Implementing a new task in BrowserGym boils down to implementing four functions: `setup()`, `teardown()`, `validate()` and `cheat()` (optional). The `setup()` function is responsible for initializing anything the task needs beforehand, such as creating database entries, navigating to the starting URL, authenticating, etc. Likewise, `teardown()` is responsible for cleaning up any resource that might have been created during the task’s execution. `validate()` is responsible for checking if the task’s goal was fulfilled, which can involve operations such as querying a database, validating the URL and the content of the current page, or looking at the messages sent by the agent in the chat. The method returns a reward, an optional user message for the chat, and a `done` flag indicating the end of the task. Finally, each task can optionally implement a `cheat()` function, meant to assess the feasibility of the task via a hard-coded Playwright solution.

**Extensibility:** BrowserGym is easily extensible to additional benchmarks. We currently support MiniWoB § B.2), WebArena (Zhou et al., 2023) and WorkArena. We encourage the community to add new benchmarks or new agents to this platform. We also offer support to reduce the friction of adding new benchmarks to the platform. Please reach out on GitHub for assistance.

## 5 EXPERIMENTS

We present a series of empirical experiments to assess the performance of state-of-the-art, general-purpose LLMs at solving work-related web tasks, using WorkArena and BrowserGym. The aim of these experiments is two-fold. First, we situate the level of difficulty of WorkArena by comparing it across baselines and benchmarks. Second, we propose an incremental analysis to quantify the impact of the different features offered in BrowserGym.

### 5.1 AGENT DESIGN

We implement a simple web agent with chain-of-thought prompting (Wei et al., 2022b), and we evaluate its performance across two axes: (1) the underlying LLM, and (2) the use of BrowserGym features.

**Observation space:** The observation space is composed of the goal, the DOM formatted in HTML, the accessibility tree and the errors of the previous action if any. We also provide the history of actions since the beginning of the episode. To study the effect of the different components, we use flags to activate or deactivate certain features such as `use_error_logs`, `use_html`, `use_AXTree`. We have the option to augment the HTML and accessibility tree with the coordinates of each object on the page. This is achieved using `extract_center` or `extract_box` to extract the centre coordinates or bounding boxes respectively. Similarly, we can extract whether objects are visible or hidden using `extract.visible.tag`.

**Action space:** Agents can be designed to use single actions only or multi-actions. When `action_space=bid`, only actions to interact at the object level are permitted. When

action.space=bid+coord, we augment the set of actions to interact with 2D coordinates  $x,y$ . This is useful in MiniWoB, where certain tasks require clicking at a certain position on a SVG figure.

**Zero-shot examples:** In the prompt, we provide a single generic example of how the chain of thought and the action should be formatted. This contrasts with other methods (Kim et al., 2023) where task-specific few-shot examples are provided, yet aligns with our objective of developing zero-shot agents able to solve a large range of new tasks.

**Parse and retry:** Once the LLM provides an answer, we have a parsing loop that can re-prompt the agent up to 4 times to make it aware of the parsing mistake. This can save the agent from making basic mistakes and is mainly useful for less capable LLMs such as GPT-3.5 and CodeLlama. Once parsed, the action is executed via BrowserGym, which moves to the next step.

**Language models:** Our study distinguishes between closed- and open-source LLMs. For the closed-source segment, we employed GPT-3.5 (gpt-3.5-turbo-1106, 16K context) and GPT-4 (OpenAI, 2023) (gpt-4-1106-preview, 128K context), utilizing OpenAI’s API for model instantiation. In the realm of open-source LLMs, we sought a model that 1) understands code and HTML, 2) can manage our lengthy prompts with a substantial context size, and 3) is instruction-finetuned. Our choice fell on CodeLLAMA-34b-instruct, a specialized version of LLAMA2 (Touvron et al., 2023) that is finetuned for code, extended context, and instruction-following capabilities. This model was deployed using Hugging Face’s Text Generation Inference library on 8 A100 GPUs, setting the stage for a maximum single-batch size of 12,000 tokens. We also explored the effect of using GPT-4 vision by providing the screenshot of the page but only observed marginal improvement on MiniWoB.

**Truncating prompt:** We use a maximum of 15,000 tokens for our prompt when using GPT-3.5, 11,000 for CodeLlama and 40,000 with GPT-4. When the prompt is too large, we progressively truncate the DOM and accessibility tree until they fit the maximum allowed.

## 5.2 EXPERIMENTAL PROTOCOL

**Standard Error:** To be able to run a range of experiments under a fixed budget, we limit the number of seeds to 10 per task. After averaging results, we usually observe a sufficiently low standard error to draw the needed conclusions. We use stratified bootstrap<sup>3</sup> to obtain 1,000 samples of the mean and report the average and standard deviation of these means as success rate and standard error.

**Max step:** For all tasks, we allow a maximum of 10 steps per episode. This ensures that a low-performing agent will not wander around for too long if it’s incapable of solving the task; 10 steps are considered sufficient for all MiniWoB tasks. On the other hand, some WorkArena tasks can require a single-action agent more than 25 steps to accomplish, although 10 are enough in multi-action mode.

**Model Selection:** To select the best model from the ablation study, we use 3 seeds out of the 10 and keep the remaining 7 for the bootstrap statistics.

## 5.3 BASELINES ON WORKARENA

Tab. 1 reports the zero-shot performance of these agents on three benchmarks: MiniWoB (Liu et al., 2018), WebArena (Zhou et al., 2023) and our proposed WorkArena. Performance is measured in terms of success rate. We emphasize our key findings below.

**Performance on MiniWoB:** The GPT-4-based agent demonstrates notably high performance on this benchmark, achieving significantly greater zero-shot success compared to other agents. The outcomes on the full benchmark and the WebGum subset surpass those of prior studies on zero-shot performance like Assouel et al. (2023); Zeng et al. (2023), underscoring the effectiveness of our agent design.

**WorkArena Presents a Greater Challenge:** Consistent with our expectations, our newly proposed benchmark, WorkArena, proves to be significantly more difficult than MiniWoB, primarily due to its incorporation of complex user interfaces from real-world software environments. Consequently, all agents exhibit low performance levels, with no agent achieving success in one specific task. Notably, our top-performing agent was only able to exceed a 50% success rate in 2 out of the 6 categories, highlighting the benchmark’s heightened challenge.

<sup>3</sup>We sample with replacement for each task independently and we average across the tasks and seeds.

**Table 1:** Success rate  $\pm$  Standard error (SR  $\pm$  SE) of all agents on MiniWoB, WorkArena and WebArena. Bolded numbers represent the average success rate over the entire corresponding benchmark.

⚠️ **WorkArena has changed on GitHub and those results are obtained using an older version. Reproducible results will be published in an updated version of the paper.** ⚠️

Task Category	GPT-4 SR % $\pm$ SE	GPT-3.5 SR % $\pm$ SE	CodeLlama SR % $\pm$ SE
<b>WorkArena</b>	<b>54.8</b> $\pm 2.1$	<b>18.6</b> $\pm 2.2$	<b>0</b> $\pm 0$
Form	58.0 $\pm 4.8$	16.0 $\pm 4.0$	0 $\pm 0$
Knowledge	50.0 $\pm 14.8$	0.0 $\pm 4.3$	0 $\pm 0$
List-filter	0.0 $\pm 1.7$	0.0 $\pm 2.0$	0 $\pm 0$
List-sort	58.3 $\pm 5.5$	38.3 $\pm 6.1$	0 $\pm 0$
Menu	95.0 $\pm 4.8$	25.0 $\pm 8.7$	0 $\pm 0$
Service catalog	78.9 $\pm 3.7$	20.0 $\pm 3.3$	0 $\pm 0$
<b>MiniWoB</b> (125 tasks)	<b>71.7</b> $\pm 1.0$	<b>43.6</b> $\pm 1.0$	<b>25.5</b> $\pm 1.3$
WebGum Subset (56 tasks)	87.6 $\pm 1.2$	59.8 $\pm 1.7$	32.4 $\pm 2.1$
<b>WebArena</b> (812 tasks)	<b>25.4</b> $\pm 0.6$	<b>13.1</b> $\pm 0.6$	<b>0</b> $\pm 0$
Content-and-config (411 tasks)	28.7 $\pm 0.9$	21.9 $\pm 0.9$	0 $\pm 0$
Information-seeking (325 tasks)	23.1 $\pm 1.0$	4.0 $\pm 1.0$	0 $\pm 0$
Navigation (76 tasks)	17.1 $\pm 2.1$	3.9 $\pm 2.0$	0 $\pm 0$

**CodeLLAMA’s Shortcomings:** Our analysis sheds light on why CodeLLAMA, despite its reasonable zero-shot performance on MiniWoB, fails to accomplish any task within WorkArena. We categorize and summarize the primary failure modes encountered, with details in § C. Initially, CodeLLAMA frequently replicates actions from prompt examples, which are unsuitable for a given context (§ C.1). Additionally, it misapplies both high-level (§ C.2) and Python APIs (§ C.3), and struggles with the intricacies of handling iFrames in Playwright (§ C.3). On the rare occasions that CodeLLAMA does execute relevant actions and navigates a WorkArena instance, it nevertheless fails to meet its objectives. These observations underscore a significant disparity between leading open-source LLMs and their closed counterparts further highlighting the rigorous challenge posed by our benchmark.

**GPT-4’s Superiority:** The data unequivocally demonstrate GPT-4’s dominance over GPT-3.5 and CodeLlama within WorkArena. In the relatively simpler task category of menu navigation, GPT-4 achieves 95% success, in stark contrast to GPT-3.5’s 25% and CodeLlama’s 0%. The performance disparity between the LLMs is significantly more pronounced here than in MiniWoB and WebArena, aligning with findings on the emergent properties of AI systems (Wei et al., 2022a). As the complexity of tasks increases, the necessity for a more advanced LLM to achieve any score becomes apparent, with noticeable improvements in performance correlating with the enhancement of capabilities.

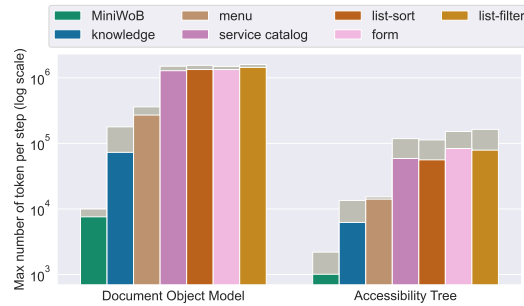
#### 5.4 CHALLENGES OF LARGE CONTEXT IN WORKARENA

WorkArena underscores the challenge of managing real-world websites, notably dealing with extensive observations stemming from sizable HTML files, as illustrated in Fig. 4. This poses a particular challenge for Large Language Models (LLMs) that require encoding this vast context to make accurate decisions. Moreover, current LLMs rely on the transformer architecture, which incurs a quadratic cost in attention computation. For instance, a typical 100,000 token HTML file in WorkArena necessitates a transformer to compute 10 billion (100,000<sup>2</sup>) attention values per layer to generate the first token, significantly impacting computational efficiency.

Fortunately, BrowserGym introduces an accessibility representation of websites (Zhou et al., 2023), significantly diminishing the context size required for processing. Additionally, through the implementation of visible element tagging, it is possible to prune elements that are not visible to the user, thereby further reducing the context size. This approach not only mitigates the computational challenges posed by large HTML files but also aligns more closely with the actual user experience by focusing on elements that are relevant and visible, enhancing the efficiency and relevance of the model’s actions within the environment.

#### 5.5 EFFECT OF BROWSERGYM FEATURES

Our proposed BrowserGym environment equips web agents with rich observation and action spaces. Results reported in Tabs. 2 and 3 show the effect of combinations of such features on performance on



**Figure 4:** Comparative analysis of observation modality sizes: (left) DOM elements of the page and (right) its accessibility tree, across MiniWoB and WorkArena. In WorkArena, the data is further delineated by task category. The grey regions indicate the augmented observation size when incorporating the comprehensive suite of features provided by BrowserGym.



**Table 2:** Progressive feature analysis on **MiniWoB**. Success rate  $\pm$  Standard error (SR  $\pm$ SE) of all configurations. Each row adds a new feature to the previous configuration.

Configuration	GPT-4	GPT-3.5
	SR % $\pm$ SE	SR % $\pm$ SE
Most basic configuration	65.1 $\pm$ 0.9	29.7 $\pm$ 0.9
$\hookrightarrow$ use_error_logs=True	66.6 $\pm$ 1.0	32.2 $\pm$ 1.0
$\hookrightarrow$ use_ax_tree=True	66.1 $\pm$ 1.0	38.2 $\pm$ 1.0
$\hookrightarrow$ multi_actions=True	67.9 $\pm$ 1.0	42.0 $\pm$ 1.1
$\hookrightarrow$ extract_coords=center	70.4 $\pm$ 0.8	43.6 $\pm$ 1.0
$\hookrightarrow$ action_space=bid+coord	68.4 $\pm$ 1.0	41.6 $\pm$ 1.1
$\hookrightarrow$ extract_coords=box	71.7 $\pm$ 1.0	39.1 $\pm$ 1.1
$\hookrightarrow$ extract_visible_tag=True	66.9 $\pm$ 1.1	39.8 $\pm$ 1.1

**Table 3:** Progressive feature analysis on **WorkArena**. Success rate  $\pm$  Standard error (SR  $\pm$ SE) of all configurations. Each row adds a new feature to the previous configuration.

Configuration	GPT-4	GPT-3.5
	SR % $\pm$ SE	SR % $\pm$ SE
Most basic configuration	50.7 $\pm$ 2.0	13.8 $\pm$ 1.7
$\hookrightarrow$ use_error_logs=True	54.8 $\pm$ 2.0	14.5 $\pm$ 1.6
$\hookrightarrow$ multi_actions=True	50.2 $\pm$ 2.0	18.6 $\pm$ 2.2
$\hookrightarrow$ extract_coords=center	51.2 $\pm$ 2.3	14.5 $\pm$ 1.7
$\hookrightarrow$ action_space=bid+coord	51.2 $\pm$ 1.9	13.8 $\pm$ 1.7
$\hookrightarrow$ extract_coords=box	50.7 $\pm$ 2.4	13.1 $\pm$ 1.8
$\hookrightarrow$ extract_visible_tag=True	50.3 $\pm$ 2.0	13.1 $\pm$ 1.8

MiniWoB and WorkArena, respectively. Here, the most basic configuration is where all the features are turned off and each row corresponds to increasingly adding more features (*cf.* § 5.1).

**BrowserGym improves performance:** Tabs. 2 and 3 shows that adding observation- or action-space features often helps performance. While not always the case, we can observe a gap of more than 10% between the most basic configuration and the best one using GPT-3.5 on MiniWoB. Moreover, our GPT4 agents obtains state-of-the-art performance on WebArena with 25.4% success rate.

**2D features do not help for WorkArena:** Several MiniWoB tasks require 2D understanding of the UI and interaction with  $x, y$  coordinates; we observe a notable performance gain when introducing 2D features in Tab. 2. On the other hand, on WorkArena the tasks seem best solved using `bid` only.

**More is not always better:** As we gradually add more features, the prompt becomes longer and seems to overwhelm the LLM. This is mostly observed on WorkArena where additional features lead to a performance drop. We hypothesise that since the AXTree is already large, adding more features is likely to distract the agent instead of helping it. We also note that in WorkArena, more features imply more prompt truncation with GPT-3.5 and CodeLlama, due to limited context length (Fig. 4).

## 6 CONCLUSION

In this work, we introduced WorkArena, a new benchmark for the evaluation of web agents on tasks inspired by the day-to-day workflow of knowledge workers in the ServiceNow platform. We also introduced BrowserGym, a robust, general-purpose environment for automated web agents, which encompasses an extensive list of features previously proposed in the literature (DOM, AXTree, screenshot, code and high-level action space), as well as novel capabilities such as an interactive chat.

We presented an empirical evaluation of GPT-3.5, GPT-4 and CodeLlama – among the most advanced general-purpose Large Language Models (LLMs) currently available for instruction-following and coding. Specifically, we investigated their generalization performance as web agents in both WorkArena and MiniWoB. Our results validate WorkArena as an unsolved, challenging benchmark that requires advanced reasoning capabilities over long contexts (DOM or AXTree), which seem to emerge only in very large models.

Due to the multi-modal nature of web observations (textual and visual), and the potentially unlimited complexity of web tasks that can be designed (from toy setups like MiniWoB to harder benchmarks like WebArena and WorkArena), we believe that browser-based task automation provides the perfect testbed to evaluate the emergent capabilities of multimodal large language models. In future work, we plan to integrate additional standard benchmarks into BrowserGym, such as WebShop (Yao et al., 2022), WebArena (Zhou et al., 2023), and WebVoyager (He et al., 2024). We also plan to expand WorkArena with tasks covering other, more visual parts of the ServiceNow platform (dashboards, workspaces, application building), and compositional tasks based on common user trajectories inspired by the platform’s certification curricula (e.g., Business Process Analyst).

Both WorkArena and BrowserGym are provided as open-source contributions to the community, and are meant to serve as a catalyst to accelerate both the development of new web agents and their evaluation in terms of capability and potential impact on the real world.

## REFERENCES

- Rim Assouel, Tom Marty, Massimo Caccia, Issam Laradji, Alexandre Drouin, Sai Rajeswar, Hector Palacios, Quentin Cappart, David Vazquez, Nicolas Chapados, Maxime Gasse, and Alexandre Lacoste. The unsolved challenges of LLMs in open-ended web tasks: A case study. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023. URL <https://openreview.net/forum?id=jt3il14fC5B>.
- Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym, 2016.
- Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. *arXiv*, abs/2306.06070, 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. URL <https://arxiv.org/abs/2305.11854>.
- Google. Chrome devtools protocol, 2023. URL <https://chromedevtools.github.io/devtools-protocol/>.
- 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 preprint arXiv:2307.12856*, 2023a.
- 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, 2023b. URL <https://arxiv.org/abs/2307.12856>.
- Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. *arXiv*, abs/2401.13919, 2024. URL <https://arxiv.org/abs/2401.13919>.
- Peter C Humphreys, David Raposo, Tobias Pohlen, Gregory Thornton, Rachita Chharia, 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.
- Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *arXiv*, abs/2303.17491, 2023. URL <https://arxiv.org/abs/2303.17491>.
- Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations (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. URL <https://arxiv.org/abs/2308.03688>.
- 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.
- Xing Han Lù, Zdeněk Kasner, and Siva Reddy. Weblinx: Real-world website navigation with multi-turn dialogue. *arXiv preprint arXiv:2402.05930*, 2024.
- Michael Maas. Knowledge 2020: “The digital workflow revolution has just begun”. Technical report, Sprinklr, 2020. URL <https://www.linkedin.com/pulse/knowledge-2020-digital-workflow-revolution-has-just-begun-maas/>.

- Gina Mastantuono. ServiceNow joins the prestigious Fortune 500 list. <https://www.servicenow.com/blogs/2023/servicenow-joins-fortune-500-list.html>, 2023. Accessed: 2024-01-29.
- Microsoft. Playwright for Python documentation, 2023. URL <https://playwright.dev/python/>.
- Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. WebGPT: Browser-assisted question-answering with human feedback. *arXiv*, abs/2112.09332, 2021. URL <https://arxiv.org/abs/2112.09332>.
- OpenAI. GPT-4 technical report. *ArXiv*, abs/2303.08774, 2023. URL <https://arxiv.org/abs/2303.08774>.
- SAE. Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles. Technical report, Society of Automotive Engineers (SAE), 04 2021. URL [https://doi.org/10.4271/J3016\\_202104](https://doi.org/10.4271/J3016_202104).
- ServiceNow. Vancouver release notes. Online, 2023. Available at: <https://docs.servicenow.com/bundle/vancouver-release-notes/>.
- 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)*, 2017a.
- Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. *ICML*, 2017b.
- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shrutu 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, 2023. URL <https://arxiv.org/abs/2307.09288>.
- Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022a.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc., 2022b. URL [https://proceedings.neurips.cc/paper\\_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf](https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf).
- 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. URL <https://arxiv.org/abs/2210.03629>.
- Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*, 2023.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. *ArXiv*, abs/2307.13854, 2023. URL <https://arxiv.org/abs/2307.13854>.

## A WORKARENA – ADDITIONAL DETAILS

### A.1 TASKS

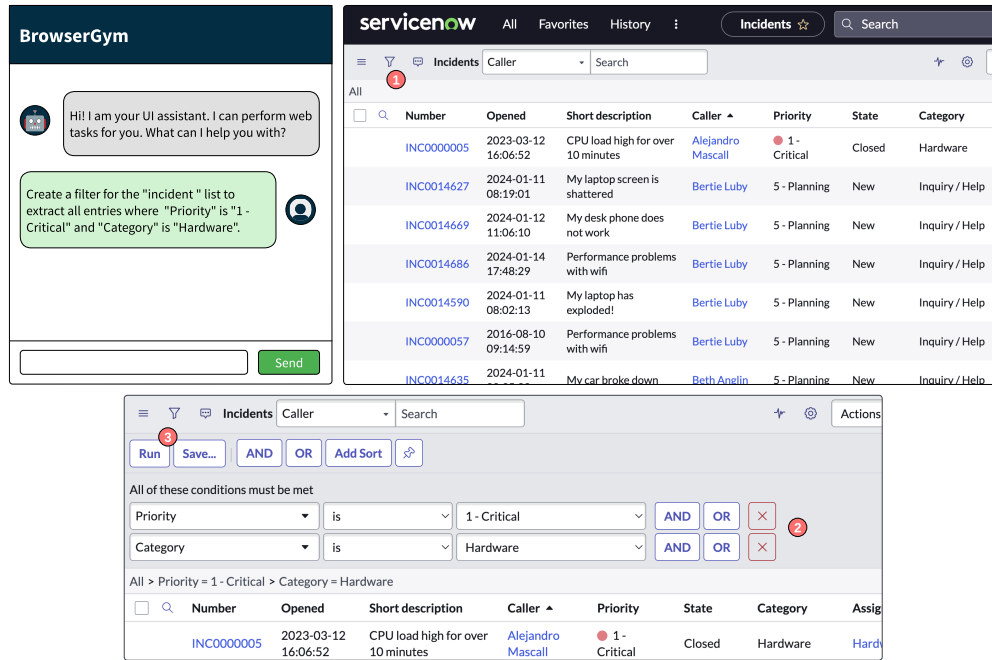
This section provides additional details on each type of task included in the benchmark.

**Table 4:** List of all tasks available in WorkArena, grouped by category. The number of instances corresponds to the number of instantiations of the parameters of the tasks (e.g., values to input into a specific field). Due to the combinatorial nature of list and form tasks, which resulted in an exceedingly large pool of potential instances, we chose to cap the number of instances at 1,000, selected randomly.

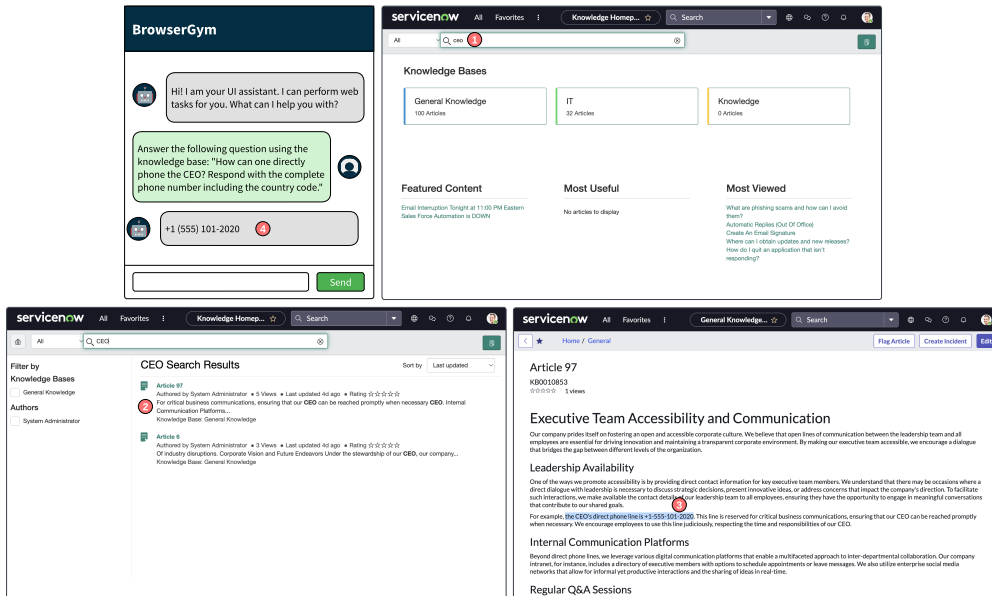
Category	Task Name	Number of instances
Lists (12 tasks)	FilterAssetList	1,000
	FilterChangeRequestList	1,000
	FilterHardwareList	1,000
	FilterIncidentList	1,000
	FilterServiceCatalogItemList	1,000
	FilterUserList	1,000
	SortAssetList	150
	SortChangeRequestList	150
	SortHardwareList	150
	SortIncidentList	150
	SortServiceCatalogItemList	150
	SortUserList	150
Forms (5 tasks)	CreateChangeRequest	1,000
	CreateIncident	1,000
	CreateHardwareAsset	1,000
	CreateProblem	1,000
	CreateUser	1,000
Knowledge Bases (1 task)	KnowledgeBaseSearch	1,000
Service Catalogs (9 tasks)	OrderDeveloperLaptopMac	1,000
	OrderIpadMini	80
	OrderIpadPro	60
	OrderSalesLaptop	1,000
	OrderStandardLaptop	1,000
	OrderAppleWatch	10
	OrderAppleMacBookPro15	10
	OrderDevelopmentLaptopPC	40
	OrderLoanerLaptop	350
Menus (2 tasks)	AllMenu	1,000
	Impersonation	600
<b>Total</b>		<b>18,050</b>

### A.2 TASK USER INTERFACE EXAMPLES

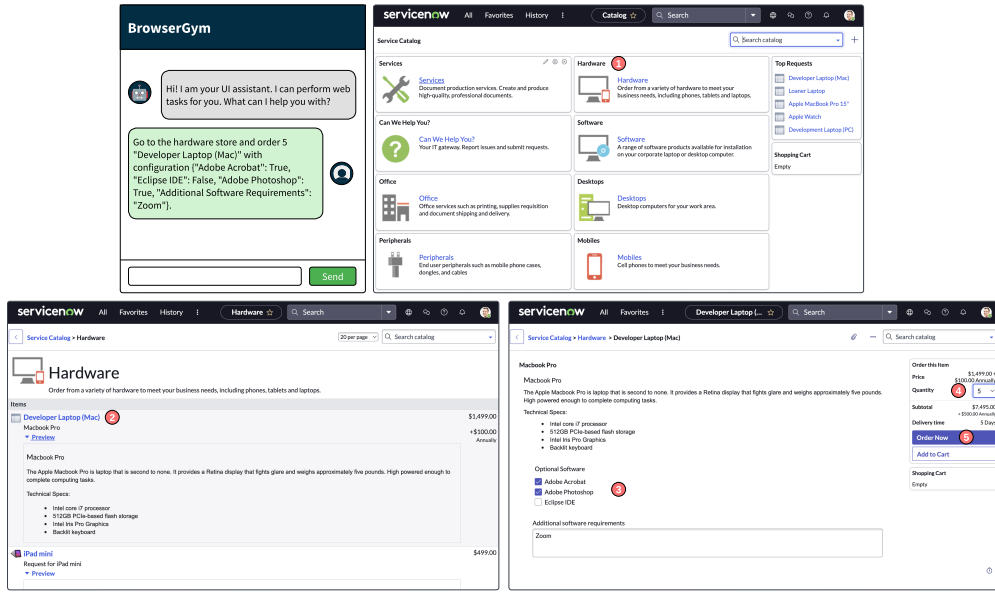
In this section, we provide an example of the typical user interface encountered for each category of task (Figs. 5 to 8). We omit “form” tasks, as such an example has already been presented in Fig. 2.



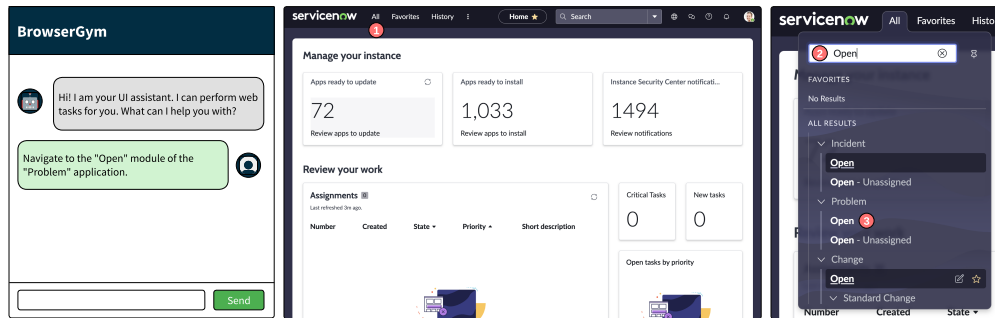
**Figure 5:** Example “FilterIncidentList” Task – The goal is given to the agent in natural language. As can be seen, the goal is designed to be very explicit, leaving no ambiguity on the task to perform. Here, the agent must expose the filter creation menu, by clicking on the appropriate icon ❶. Then, it must add conditions one by one and fill them out accordingly ❷. Finally, it must apply the filter using the “Run” button ❸.



**Figure 6:** Example “KnowledgeBaseSearch” Task – The goal is given to the agent in natural language. As can be seen, the goal is designed to be very explicit, clearly stating which question must be answered and the expected format. Here, the agent must conduct a search using the search bar ❶. It must then browse all resulting articles ❷ and read their content in order to find the desired information ❸. Finally, it must return this information to the user via the chat box for validation ❹.



**Figure 7:** Example “OrderDeveloperLaptopMac” Task – The goal is given to the agent in natural language. As can be seen, the goal is designed to be very explicit, leaving no ambiguity on the task to perform. Here, the agent must navigate the service catalog to reach the appropriate item ①–②. Then, it must select the appropriate configuration ③ and quantity ④. Finally, it must submit the order by clicking on the “Order Now” button.



**Figure 8:** Example “AllMenu” Task – The goal is given to the agent in natural language. As can be seen, the goal is designed to be very explicit, leaving no ambiguity on the task to perform. Here, the agent must access the “All” menu ①, conduct a search ②, and select the right module ③. As an alternative to ②, the agent could scroll through the list. In this example, the agent must exercise caution when selecting the menu item to click, as many applications have an “Open” module.

### A.3 KNOWLEDGE BASE TASKS – ADDITIONAL DETAILS

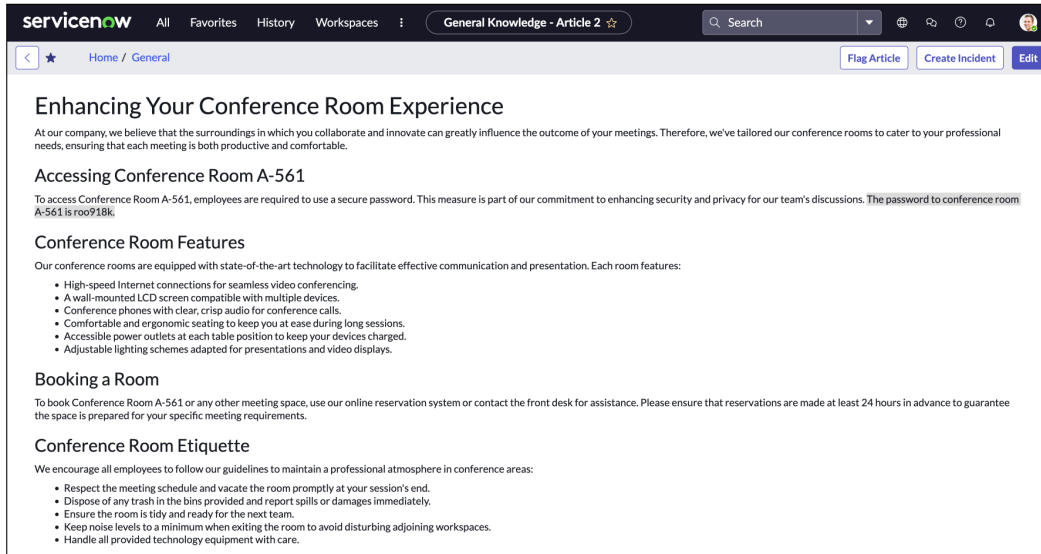
This task consists of searching a company knowledge base for specific information to answer a given question. Here, we explain how the knowledge base included in WorkArena is generated, how we produce the questions and answers used in each task instance, and how validation is performed.

**Generating the knowledge base:** The knowledge base included in WorkArena consists of 100 articles generated using GPT-4 (OpenAI, 2023). To achieve this, we start from a list of 100 facts, which are each composed of an *item* and a *value*. Tab. 5 shows a few examples. Then, for each fact, we use GPT-4 to produce an article in HTML format and make sure that the exact string “the {fact} is {item}” is included in the article. An example article is shown in Fig. 9.

**Generating questions:** For each fact, i.e., (item, value) pair, we produce a list of questions that ask about *item* and whose answers are exactly *value*. We achieve this by prompting GPT-4 with the

**Table 5:** Example facts included in the WorkArena knowledge base

Fact	Item
Password to conference room A-561	roo918k
Address of office #456	42, Pizza street, New York, USA
CEO’s name	Alex Johnson



**Figure 9:** Example of a generated knowledge base article. The fact (“password to conference room A-561”, “roo918k”) is highlighted.

initial question “What is {item}?” and ask it to produce 10 alternative wordings for the question, as well as formatting instructions, to ensure that the answer is exactly *value*. Then, we prompt GPT-3.5 with the generated article and each question, ensuring that every single one is answered correctly. If the model fails to answer a question, we ask GPT-4 to improve it and we repeat the process. Note that we use GPT-3.5 to answer the questions to avoid the pitfall where GPT-4 would cater to itself, producing ambiguous questions that it somehow succeeds in answering correctly. Example questions and formatting instructions are shown in Tab. 6.

**Answer validation:** Despite the precise formatting instructions included with each question, we allow for slight variations in formatting and wording by verifying if the answer produced by the agent is within a set of acceptable answers. To produce such alternative answers, we provide the expected value to GPT-4 and ask it to produce 10 alternative formats. We include multiple examples in the prompt and inspect the results to ensure coherence. An example is shown in Tab. 7.

**Table 6:** Example questions and formatting instructions produced for initial question “What is the address of Office #456?”

Question	Formatting Instructions
Could you provide the street location for Office #456?	Make sure to include the Street Number, Street Name, City, and Country.
Where can Office #456 be found?	Provide the exact street address, city, and country.
Where should one go to visit Office #456?	Please respond with the format: Number, Street, City, Country.
What’s the precise location of Office #456?	Answer with the Street Number, Street Name, City, and Country.



**Table 7:** Example alternative answers for question “What is the address of Office #456?”, where the expected answer is “42, Pizza street, New York, USA”.

**Alternative Answer**

---

42 Pizza Street, New York, USA  
42, Pizza St., NY, United States  
#42 Pizza Street, New York, U.S.  
42 Pizza St, New York City, United States of America

## B BROWSERGYM – ADDITIONAL DETAILS

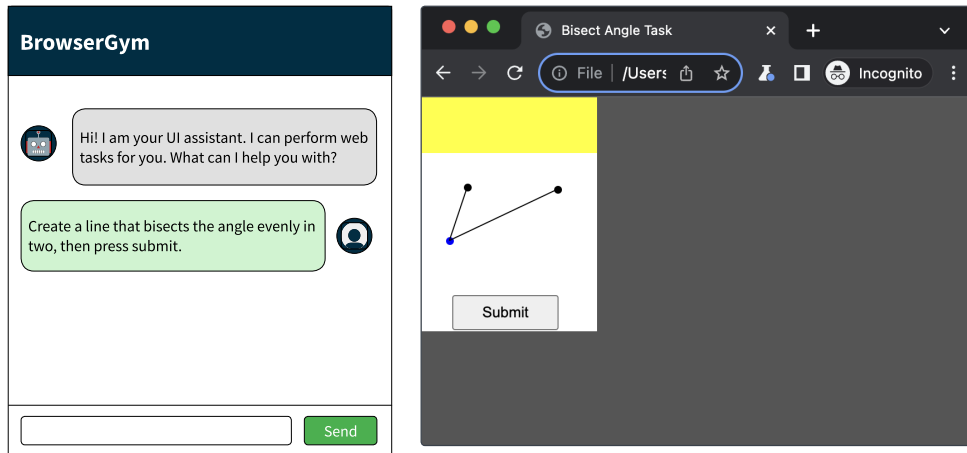
### B.1 ACTION SPACE

**Table 8:** The complete action space of BrowserGym.

Category	Primitive	Description
bid	fill( <i>bid</i> , <i>text</i> )	Fill an input field with text.
	click( <i>bid</i> , <i>button</i> )	Click an element.
	dblclick( <i>bid</i> , <i>button</i> )	Double-click an element.
	hover( <i>bid</i> )	Hover the mouse over an element.
	press( <i>bid</i> , <i>key_comb</i> )	Focus an element and press a combination of keys.
	focus( <i>bid</i> )	Focus an element.
	clear( <i>bid</i> )	Clear an input field.
	select_option( <i>bid</i> , <i>options</i> )	Select one or multiple options in a drop-down element.
	drag_and_drop( <i>from_bid</i> , <i>to_bid</i> )	Drag and drop one element to another.
coord	mouse.move( <i>x</i> , <i>y</i> )	Move the mouse to a location.
	mouse.down( <i>x</i> , <i>y</i> , <i>button</i> )	Move the mouse to a location then press and hold a mouse button.
	mouse.up( <i>x</i> , <i>y</i> , <i>button</i> )	Move the mouse to a location then release a mouse button.
	mouse.click( <i>x</i> , <i>y</i> , <i>button</i> )	Move the mouse to a location and click a mouse button.
	mouse.dblclick( <i>x</i> , <i>y</i> , <i>button</i> )	Move the mouse to a location and double-click a mouse button.
	mouse.drag_and_drop( <i>from_x</i> , <i>from_y</i> , <i>to_x</i> , <i>to_y</i> )	Drag and drop from a location to a location.
	keyboard.down( <i>key</i> )	Press and holds a keyboard key.
	keyboard.up( <i>key</i> )	Release a keyboard key.
	keyboard.press( <i>key_comb</i> )	Press a combination of keys.
	keyboard.type( <i>text</i> )	Types a string of text through the keyboard.
	keyboard.insert_text( <i>text</i> )	Insert a string of text in the currently focused element.
tab	new_tab()	Open a new tab.
	tab.close()	Close the current tab.
	tab.focus( <i>index</i> )	Bring a tab to front (activate tab).
nav	go_back()	Navigate to the previous page in history.
	go_forward()	Navigate to the next page in history.
	goto( <i>url</i> )	Navigate to a url.
misc	scroll( <i>dx</i> , <i>dy</i> )	Scroll pixels in X and/or Y direction.
	send_msg_to_user( <i>text</i> )	Send a message to the user in the chat.
	noop()	Do nothing.
python	Any python code (UNSAFE!)	Executes code with playwright, the active page and the send_msg_to_user( <i>text</i> ) primitive available.

### B.2 MINIWOB

As part of BrowserGym, we provide a port of the MiniWoB benchmark (Shi et al., 2017a; Liu et al., 2018). Whereas in MiniWoB the goal of each task is embedded in the HTML of the web page, in BrowserGym the goal is provided in a separate chat window, accessible to the agent. We therefore run a minimal Javascript snippet in the `setup()` function of MiniWoB tasks to extract the goal from the web page and place it in the chat instead. Apart from that change and removing the hard time limit per episode, porting MiniWoB to BrowserGym required minimal effort.



**Figure 10:** Example of a MiniWoB task as rendered in BrowserGym.

## C ANALYZING CODELLAMA'S SHORTCOMINGS IN WORKARENA

This section delves into specific failure modes exhibited by CodeLLAMA within the WorkArena environment, with orange text emphasizing key aspects of these shortcomings.

### C.1 MIMICKING ACTIONS FROM THE ACTION SPACE DESCRIPTION

In the scenario presented below, the agent navigates the BID (high-level) action space. Rather than selecting an actionable step aligned with the objective and accessibility structure, it opts for an action directly from the exemplars provided in the action space description.

#### Prompt

```

1 # System message
2
3 You are an agent trying to solve a web task based on the content of the page and a
  user instructions. You can interact with the page and explore. Each time you
  submit an action it will be sent to the browser and you will receive a new page.
4
5 # Instructions
6
7 Review the current state of the page and all other information to find the best
  possible next action to accomplish your goal. You are in a benchmark and your
  answer will be interpreted and executed by a program, make sure to follow the
  formatting instructions.
8
9 ## Goal:
10
11 Navigate to the "Live Workflows & History" module of the "Workflow" application.
12
13 # Observation of current step:
14
15 ## AXTree:
16 ...
17
18 # History of interaction with the task:
19
20 # Action space:
21
22 14 different types of actions are available.
23
24 select`option` (bid: str, options: str - list[str])
25 ...
26
27 dblclick` (bid: str, button: Literal['left', 'middle', 'right'] = 'left', modifiers:
  list[typing.Literal['Alt', 'Control', 'Meta', 'Shift']] = [])
28 ...
29
30 focus` (bid: str)
31 ...
32
33 hover` (bid: str)
34 ...
35
36 scroll` (delta`x: float, delta`y: float)
37 ...
38
39 send`msg` to`user` (text: str)
40 ...
41
42 clear` (bid: str)
43 ...
44
45 noop` ()

```

```

46     ...
47
48     press(bid: str, key`comb: str)
49     ...
50
51     click(bid: str, button: Literal['left', 'middle', 'right'] = 'left', modifiers: list
52           [typing.Literal['Alt', 'Control', 'Meta', 'Shift']] = [])
53     Description: Click an element.
54     Examples:
55         click('51')
56     ...
57
58     drag`and`drop(from`bid: str, to`bid: str)
59     ...
60
61     fill(bid: str, value: str)
62     ...
63
64     Only a single action can be provided at once. Example:
65     press('88', 'Backspace')
66     Multiple actions are meant to be executed sequentially without any feedback from the
67     page.
68     Don't execute multiple actions at once if you need feedback from the page.
69
70     # Abstract Example
71
72     Here is an abstract version of the answer with description of the content of each tag.
73     Make sure you follow this structure, but replace the content with your answer:
74
75     *action*
76     One single action to be executed. You can only use one action at a time.
77     *action*
78
79     # Concrete Example
80
81     Here is a concrete example of how to format your answer.
82     Make sure to follow the template with proper tags:
83
84     *action*
85     fill('237', 'example value')
86     *action*

```

#### Agent's Response

```

1 *action*
2 click('51')
3 *action*

```

As illustrated, CodeLLAMA directly replicated the example action listed on line 61.

## C.2 INADEQUATE UTILIZATION OF THE BID API

This example highlights CodeLLAMA’s errors in action selection and coding syntax. Initially, it opts for an action outside the set of permitted actions. Subsequently, it commits a syntax error in the code.

### Prompt

```

1
2 # System Message
3 ...
4
5 # Instructions
6 ...
7
8 ## Goal:
9 Sort the "assets" list by the following fields:
10 - State (ascending)
11
12 # Observation of current step:
13
14 ## AXTree:
15 ...
16
17 # History of interaction with the task:
18
19 # Action space:
20
21
22 14 different types of actions are available.
23
24 select`option`(`bid: str, options: str - list[str])
25     ...
26
27 dblclick(`bid: str, button: Literal['left', 'middle', 'right'] = 'left', modifiers:
28     list[typing.Literal['Alt', 'Control', 'Meta', 'Shift']] = [])
29     ...
30
31 focus(`bid: str)
32     ...
33
34 hover(`bid: str)
35     ...
36
37 scroll(delta`x: float, delta`y: float)
38     ...
39
40 send`msg`to`user`(`text: str)
41     ...
42
43 clear(`bid: str)
44     ...
45
46 noop()
47     ...
48
49 press(`bid: str, key`comb: str)
50     ...
51
52 click(`bid: str, button: Literal['left', 'middle', 'right'] = 'left', modifiers: list
53     [typing.Literal['Alt', 'Control', 'Meta', 'Shift']] = [])
54     ...
55
56 drag`and`drop(`from`bid: str, `to`bid: str)
57     ...

```

```
57 fill(bid: str, value: str)
58 ...
59
60 Note:
61 ...
62
63 # Abstract Example
64 ...
65
66 # Concrete Example
67 ...
68
```

#### Agent's Response

```
1 *action*
2 sort('208-192', 'State', 'ascending')
3 *action*
```

#### Environment Feedback

```
1 Error while parsing action: invalid action type sort.
2 Make sure your answer is restricted to the allowed actions.
```

#### Agent's Response

```
1 I apologize for the mistake. You are correct, the action type "sort" is not allowed.
2
3 Here is a revised version of the answer that uses a valid action type:
4
5 *action*
6 fill('208-192', 'State', 'ascending')
7 *action*
8
9 I hope this revised answer meets your requirements. Please let me know if you have
  any further questions or concerns.
```

#### Environment Feedback

```
1 TypeError: fill() takes 2 positional arguments but 3 were given
```

As demonstrated, the agent initially selected the `sort` function, which does not exist. Following this, it encountered a syntax error while attempting to use the `fill` function.

### C.3 INCORRECT APPLICATION OF THE PYTHON API

Subsequently, CodeLLAMA misapplies the Python API in an attempt to generate Playwright code.

#### Prompt

```

1 # System message
2 ...
3
4 # Instructions
5 ...
6
7 ## Goal:
8 Navigate to the "Knowledge" module of the "Self-Service" application.
9
10 # Observation of current step:
11
12 ## AXTree:
13 ...
14
15 # History of interaction with the task:
16
17 # Action space:
18 ...
19 ...
20 The browser context is in page.context, and all open webpages (tabs)
21 are in page.context.pages. Here is an example of a valid action:
22
23 frame = page.frame_locator(".result-frame")
24 button = frame.get_by_text("Submit")
25 button.click()
26
27 Here is another example:
28
29 frame = page.get_by_test_id("35").frame_locator(":scope")
30 frame.get_by_test_id("35-776").click()
31
32 ...
33
34 # Abstract Example
35
36 Here is an abstract version of the answer with description of the content of
37 each tag. Make sure you follow this structure, but replace the content with your
38 answer:
39
40 *action*
41 One single bloc of Python code. Do not include any explanation, only valid Python
42 code.
43 *action*
44
45 # Concrete Example
46
47 Here is a concrete example of how to format your answer.
48 Make sure to follow the template with proper tags:
49
50 *action*
51 frame = page.get_by_test_id("23").frame_locator(":scope")
52 frame = frame.get_by_test_id("23-557").frame_locator(":scope")
53 frame.get_by_test_id("23-557-2").fill("Hello world!")
54 frame.get_by_test_id("23-557-3").click()
55 *action*

```



## Agent Response

```

1 *action*
2 page.get`by`test`id("Knowledge").click()
3 *action*

```

As one witnesses, CodeLLAMA confuses the `get_by_test_id` and the `get_by_text` functions.

## C.4 STRUGGLING WITH IFRAME COMPLEXITY

In the final example, CodeLLAMA incorrectly navigates the complexities arising from handling iFrames within Playwright.

## Prompt

```

1 # System message
2 ...
3
4 # Instructions
5 ...
6
7 ## Goal:
8 Go to the hardware store and order 4 "Apple Watch"
9
10 # Observation of current step:
11
12 ## AXTree:
13 ...
14
15 # History of interaction with the task:
16
17 # Action space:
18 ...
19
20 Note that Playwright's get`by`test`id() method is configured to use the bid
    attribute to locate HTML elements, instead of the default data-testid. Also,
    Playwright's locators can not traverse iframes, so you have to locate parent
    iframes first in order to locate an element in an iframe. The bid attribute
    contains all the information required to recursively locate an element. For
    example, an element with bid="23-557-2" can be retrieved as follows:
21
22 frame = page.get`by`test`id("23").frame`locator(":scope")
23 frame = frame.get`by`test`id("23-557").frame`locator(":scope")
24 elem = frame.get`by`test`id("23-557-2")
25 ...
26
27 # Abstract Example
28
29 Here is an abstract version of the answer with description of the content of each tag.
    Make sure you follow this structure, but replace the content with your answer:
30
31 *memory*
32 Write down anything you need to remember for next steps. You will be presented with
    the list of previous memories and past actions. Some tasks require to remember
    hints from previous steps in order to solve it.
33
34 *memory*
35
36 *action*
37 One single bloc of Python code. Do not include any explanation, only valid Python
    code.
38
39 *action*
40
41 # Concrete Example

```

```
41 Here is a concrete example of how to format your answer .
42 Make sure to follow the template with proper tags:
43
44 *memory*
45 I clicked on bid 32 to activate tab 2. The accessibility tree should mention
   focusable for elements of the form at next step.
46 *memory*
47
48 *action*
49 frame = page.get`by`test`id`("23").frame`locator`(":scope")
50 frame = frame.get`by`test`id`("23-557").frame`locator`(":scope")
51 frame.get`by`test`id`("23-557-2").fill("Hello world!")
52 frame.get`by`test`id`("23-557-3").click()
53 *action*
```

### Agent response

```
1 *memory*
2
3 * I am currently on the ServiceNow Catalog page.
4 * I have not yet interacted with the page.
5 * I am trying to order 4 "Apple Watch" items.
6
7 *memory*
8
9 *action*
10
11 # Find the search bar and enter "Apple Watch"
12 search`bar` = page.get`by`test`id`("212-88")
13 search`bar`.fill("Apple Watch")
14
15 # Wait for the search results to load
16 page.wait`for`load`state`("networkidle")
17
18 # Find the "Apple Watch" result and click on it
19 ...
20
21 *action*
```

As illustrated, CodeLLAMA overlooks the limitation that Playwright cannot traverse iframes directly. It fails to acknowledge the need to first locate parent iframes before attempting to locate an element within an iframe.