Web Agents can Self-Improve by Discovering and Honing Skills

Boyuan Zheng[1]*, **Michael Y. Fatemi**[2]*, **Xiaolong Jin**[3]*,
Zora Zhiruo Wang[4], **Apurva Gandhi**[4], **Yueqi Song**[4], **Yu Gu**[1], **Jayanth Srinivasa**[5], **Gaowen Liu**[5],
Graham Neubig[4], **Yu Su**[1]

[1] The Ohio State University [2] University of Virginia [3] Purdue University
[4] Carnegie Mellon University [5] Cisco Research
{zheng.2372, su.809}@osu.edu

Abstract

To survive and thrive in complex environments, humans have evolved sophisticated self-improvement mechanisms through environment exploration, hierarchical abstraction of experiences into reuseable skills, and collaborative construction of an ever-growing skill repertoire. Despite recent advancements, autonomous web agents still lack crucial self-improvement capabilities, struggling with procedural knowledge abstraction, refining skills, and skill composition. In this work, we introduce SKILLWEAVER, a skill-centric framework enabling agents to self-improve by autonomously synthesizing reusable skills as APIs. Given a new website, the agent autonomously discovers skills, executes them for practice, and distills practice experiences into robust APIs. Iterative exploration continually expands a library of lightweight, plug-and-play APIs, significantly enhancing the agent's capabilities. Experiments on WebArena and real-world websites demonstrate the efficacy of SKILLWEAVER, achieving relative success rate improvements of 31.8% and 39.8%, respectively. Additionally, APIs synthesized by strong agents substantially enhance weaker agents through transferable skills, yielding improvements of up to 54.3% on WebArena.

1 Introduction

AI agents based on large language models (LLMs) that can browse the web (Deng et al., 2023; Zhou et al., 2024a; Zheng et al., 2024) or use computers (Xie et al., 2024) like humans are rapidly rising as a new frontier of AI research and application. Despite these promising opportunities, digital environments present substantial challenges due to their inherent complexity and diversity. Website environments are highly intricate, consisting of numerous interactive elements that create large action spaces. An even greater challenge lies in developing generalist web agents capable of generalizing to out-of-distribution task types and adapting to novel websites. Existing efforts have attempted to train web agents using large-scale trajectory datasets collected across diverse websites and task types (Li et al., 2024; Pahuja et al., 2025). However, these agents often struggle with overfitting to specific website structures and task distributions (Li et al., 2024; Zheng et al., 2024), reducing their ability to handle previously unseen environments effectively.

As an integral feat of human intelligence, self-improvement presents a promising solution to these challenges. Imagine a user visiting 'yelp.com' for the first time. In the beginning, she may not be familiar with the layout and functionalities provided by the website, so she would explore the website and think about what tasks can be done. As she does more tasks on the website (*e.g.*, searching for restaurants with various filters), common routines become memorized *procedural knowledge* (*e.g.*, searching for high-end Korean restaurants entails three steps: type '*Korean*' in the top search bar, click the '*Search*' button shaped like a magnifier,

*Equal Contribution.

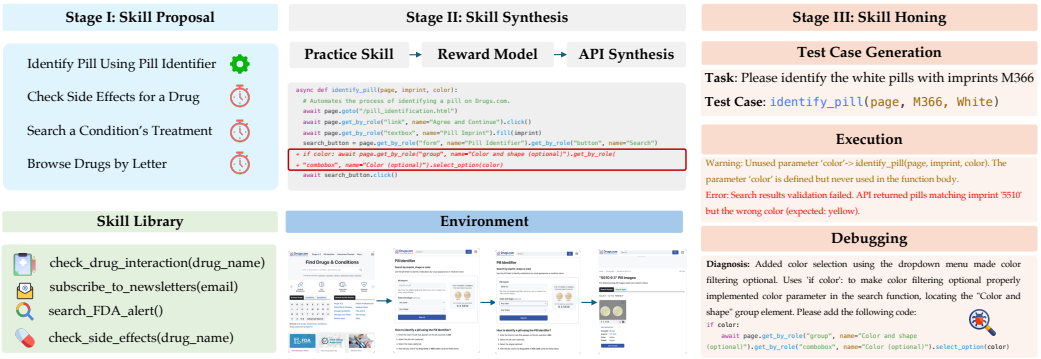

Figure 1: An overview of the SKILLWEAVER framework. The Skill Proposal module (Stage I) identifies novel skills to practice based on observations of the environment and available APIs in the skill library. For each proposed skill, the agent executes it to generate trajectories, which are later evaluated by the reward model. If successful, the trajectory is utilized to synthesize an API (Stage II). To ensure robustness, the synthesized API undergoes testing with automatically generated test cases and debugging within the Skill Honing module (Stage III).

and click the '$$$$' button on the search result page). Such routines abstracted from experiences essentially become *high-level actions* or *skills* that can be effortlessly summoned and carried out instead of re-learning things on the fly; they can even be composed into more complex routines, *e.g.*, making a reservation at a restaurant involves first finding it. Similarly, through self-improvement, agents can build conceptual maps of the website environment, accumulate procedural knowledge as skills, compose simple skills into more complex ones, and leverage these learned skills to enhance their decision-making processes.

A few recent studies have tried to capture some facets of humans' self-improvement capability. Traditional methods typically store skills implicitly through action trajectories, primarily leveraging them as demonstrations for in-context learning (Murty et al., 2024b) or fine-tuning (Murty et al., 2024a; Su et al., 2025; Pahuja et al., 2025). Although these trajectory-based approaches can be effective, they struggle to explicitly abstract reusable procedural knowledge, resulting in heavy training demands and limited generalization to new websites and tasks. Furthermore, continuously updating models with new trajectories introduces significant concerns such as catastrophic forgetting and sensitivity to website changes. Additionally, storing and sharing extensive memory-intensive trajectory data also poses practical challenges for knowledge transfer among agents.[1] Efforts like Agent Workflow Memory (Wang et al., 2024e) and ICAL (Sarch et al., 2024) take this a step further by generating abstract, reusable routines. However, both methods focus on online learning with access to test queries or offline learning requiring high-quality annotated demonstrations. Instead, we focus on more challenging autonomous exploration where agents propose novel tasks without external supervision.

To this end, we propose SKILLWEAVER, a skill-centric framework that enables web agents to autonomously self-improve by exploring website environments and synthesizing reusable, structured skills (§ 2). Our framework leverages the ability of LLMs to iteratively curate APIs based on feedback from the interactive environment (Wang et al., 2023a; 2024d). Specifically, SKILLWEAVER comprises a three-stage pipeline (Figure 1): (1) systematic exploration of website environments to identify potential skills, (2) practicing and converting these skills into robust, reusable APIs, and (3) testing and debugging to ensure API reliability during inference.

Results on WebArena (Zhou et al., 2024a) demonstrate substantial performance improvements after equipping web agents with APIs synthesized by SKILLWEAVER. We observe rel-

[1]For example, screenshot and HTML files consume approximately 0.3 GB per trajectory with an average of 7.3 actions in Mind2Web (Deng et al., 2023), making them hard to transfer on the fly.

ative gains in success rate from 25% to 38% with just 160 iterations of the three-stage pipeline. Further experiments with weaker agents revealed even more dramatic improvements, ranging from 40% to 130%. These results underscore two insights: (1) synthesized APIs can substantially enhance agent performance, and (2) weaker agents particularly benefit from APIs generated by more powerful counterparts, effectively distilling advanced knowledge into plug-and-play modules. To further validate the practicality of our method, we conduct experiments on real-world websites using tasks sampled from Online-Mind2Web (Xue et al., 2025; Deng et al., 2023). SKILLWEAVER achieved a 39.8% relative improvement in success rate, demonstrating its effectiveness in complex, dynamic web environments.

2 SkillWeaver

Our objective is to develop a framework that enables web agents to autonomously discover skills and construct a continuously expanding skill library in the form of APIs. In our work, "API" refers to a Python function containing Playwright code for a browser automation, rather than a (for example) REST API that handles HTTP requests. As illustrated in Figure 1, the exploration pipeline consists of three key modules: (1) **Skill Proposal**: Identifying new skills (*e.g.*, a short description such as *"Identify pill using pill identifier"*) for the agent to practice, facilitating the continuous discovery of relevant functionalities; (2) **Skill Synthesis**: Practicing the proposed skills and converting practiced trajectories into reusable Python functions, encapsulating learned behaviors into structured APIs; (3) **Skill Honing**: Testing and debugging the synthesized APIs using feedback from the environment and the reward model to ensure reliability and effectiveness.

2.1 Stage I: Skill Proposal

Website environments encompass numerous underlying functionalities that require systematic exploration. LLMs have been employed as automatic curriculum (Tajwar et al., 2025) for exploring open-ended environments, like Minecraft (Wang et al., 2023a; Du et al., 2023), TextWorld (Song et al., 2024a), and household settings (Du et al., 2023). Similarly, we leverage LLMs as an automatic curriculum for discovering website functionalities, leveraging their internet-scale knowledge and HTML interpretation capabilities. To facilitate the understanding of website environments, we provide LLMs with detailed webpage observations, including screenshots, website names, URLs, and accessibility trees. Accessibility trees are particularly useful for understanding less familiar websites, such as CMS in WebArena, because they provide comprehensive structural and functional information.

In contrast to existing approaches (Zhou et al., 2024b; Murty et al., 2024b;a; Song et al., 2024a), we emphasize skill diversity by explicitly prompting LLMs to propose novel and reusable skills beyond the current skill repertoire. To ensure efficient exploration and high success rates in subsequent skill practice and API synthesis stages, we specifically target short-horizon, reusable skills that can be completed within a single API call. Our approach follows a curriculum progression from simple to complex skill compositions. The LLM is instructed to propose the following three types of tasks (Prompts in Appendix § B.1):

Procedural Tasks. Procedural tasks require a sequence of atomic actions to achieve higher-level process automation objectives. Each procedural task corresponds to a workflow comprising actions that can be generalized to complete similar tasks. For example, the task *identifying pills based on imprint and color* involves a workflow where values are entered into textboxes, followed by clicking a submit button to finalize the search operation.

Navigational Tasks. Navigational tasks involve systematically exploring various sections or pages within a website. Through these tasks, agents can construct conceptual maps that capture the functionality of various webpages within the website. Examples include *navigating to the "customer reviews" section* on a product management website and *accessing individual user profiles*.

Information-Seeking Tasks. Information-seeking tasks involve scraping detailed data from webpages (*e.g.*, extracting all commits from a GitHub repository). Enumerating all available items on webpages, such as comprehensive lists of reviews or faculty members, is typically

challenging and requires extensive, repetitive actions. To address this challenge, generating specialized scraping APIs enables agents to efficiently retrieve all relevant information.

2.2 Stage II: Skill Synthesis

This module aims at automatically generating robust and reusable APIs, consisting of the following three components: Skill Practice, Reward Model, and API Synthesis. A critical challenge in this process is ensuring the robustness of the generated APIs, as each individual module may introduce errors. We address this challenge through curriculum progression—advancing from simple to complex skill compositions—which improves accuracy since modules perform better on simpler tasks requiring fewer steps. Additionally, the subsequent Skill Honing module (Stage III) validates APIs through unit testing and debugging to ensure reliability.

Skill Practice. Given a task proposed in Stage I, the agent takes actions to complete it. For procedural and navigational tasks, we leverage a base agent to generate actions to repeatedly attempt to complete tasks. The agent repeatedly attempts to complete the tasks proposed in the previous section, using the reward model to determine successful completion. For information-seeking tasks, we synthesize data extraction code to gather target information from webpages, incorporating any necessary navigation steps, such as retrieving historical orders or customer information.

Reward Model. LLMs have demonstrated effectiveness as evaluators of correctness, both for answer accuracy (Zheng et al., 2023) and for determining whether action sequences successfully complete task instructions (Pan et al., 2024; Zhuge et al., 2024). We prompt an LLM to provide reward signals indicating task completion success with the following observations in the context: (1) Task description, (2) Action trajectory comprising a sequence of actions with screenshots and corresponding descriptions, (3) Environmental feedback, including code execution outcomes and observable website changes. Further details can be found in Appendix § B.5.

API Synthesis. This module encapsulates the agent's actions from successful trajectories into a reusable and generalized API. This is performed by creating a string representation of each of the state-action pairs in the trajectory and prompting the language model to generate a Python implementation. We statically analyze the function for common generation mistakes, as described in Appendix § B.6, and if any are detected, we prompt the model to generate the function again. As illustrated in Figure 2, each API includes a Python implementation (including a function signature, docstring, and code body), where the docstring particularly contains a usage log recording previous executions of the function and a description of the prerequisite state of the website for the function to execute without errors.

```python
async def identify_pill(page, imprint, color=None, shape=None):
    """
    Automates the process of identifying a pill using the Pill Identifier
    feature on Drugs.com.

    Parameters:
    - page: The Playwright page object.
    - imprint: The imprint on the pill to be identified.
    - color: (Optional) The color of the pill.
    - shape: (Optional) The shape of the pill.

    This function navigates to the Pill Identifier page, agrees to the terms,
    inputs the pill's characteristics,
    and submits the information for identification.

    Usage Log:
    - Successfully navigated to the Pill Identifier page and submitted pill
    information for identification.
    - Inputted imprint '93 5510', color 'White', and shape 'Oval' and
    successfully submitted for identification.
    - Encountered issues with strict mode violations when attempting to click
    the 'Search' button due to multiple matches.
    - Updated to use a more specific selector for the 'Search' button to avoid
    strict mode violations.
    """
    import re

    await page.goto("https://www.drugs.com/pill_identification.html")
    await page.get_by_role("link", name="Agree and Continue").click()
    await page.get_by_role("textbox", name="Pill Imprint").fill(imprint)
    if color:
        await page.get_by_role("group", name="Color and shape (optional)").
            get_by_role("combobox", name="Color
                (optional)").select_option(color)
    if shape:
        await page.get_by_role("group", name="Color and shape (optional)").
            get_by_role("combobox", name="Shape
                (optional)").select_option(shape)

    search_button =
           page.locator("button.ddc-btn.ddc-btn-block[data-submit-loading]")
    await search_button.click()
```

Figure 2: An example of a synthesized API from Drugs.com used to identify pills based on their characteristics.

2.3 Stage III: Skill Honing

Despite significant efforts within the above modules to ensure the robustness of the synthesized API, it is not always guaranteed. To address this limitation, we integrate a stage to test and debug synthesized APIs. For APIs requiring no extra parameters except the by default Playwright page instance, we execute the API directly as a standalone unit test. For APIs requiring additional parameters, we leverage the LLM to generate appropriate parameter values that serve as comprehensive test cases.

3 Experiments

We conducted experiments on both WebArena (Zhou et al., 2024a) and real-world live websites to evaluate our agent. WebArena is a widely used benchmark for web agent evaluation, providing a diverse set of websites designed to simulate real-world web interactions. Real-world websites further demonstrate the performance of our method in more diverse, dynamic, and complex website environments. We perform exploration on each website environment separately to derive the API library, as described in Section § 3.4.

3.1 WebArena

WebArena is a self-hostable, sandboxed web environment designed for developing and evaluating web agents. It simulates real-world websites across five common application domains: e-commerce (Shopping), social forums (Reddit), collaborative software development (Gitlab), content management (CMS), and navigation (Map). This benchmark includes a total of 812 tasks, each with automated evaluation metrics based on functional correctness. WebArena provides a stable testing environment with reliable automatic evaluation, making it a good platform for demonstrating the efficacy of our agent and analyzing various aspects of our approach. Further details on the WebArena benchmark used in our evaluation can be found in Appendix § C.

Human-Crafted APIs. The self-hosted WebArena sandbox environment grants access to its source code and administrative access for the simulated websites. This enables us to extract APIs that adhere to standardized protocols (*e.g.*, REST) from the official documentation within the source code as well as from external sources (Song et al., 2024a). We further leverage these human-crafted official APIs to compare them against the APIs synthesized by SKILLWEAVER, demonstrating the quality of the generated APIs.

3.2 Real-World Websites

Real-world websites present greater complexity and richness than simulated environments. To demonstrate the practicality and efficacy of our approach in real-world environments, we conduct evaluations on live websites. Specifically, we use Online-Mind2Web (Xue et al., 2025; Deng et al., 2023), a benchmark specifically designed for evaluating web agents on live websites. This benchmark encompasses 300 tasks across 136 popular websites spanning diverse domains. The tasks represent realistic scenarios commonly encountered in daily web interactions, as proposed by human annotators. Considering the cost of exploration on all the websites, we only consider websites with a sufficient number of tasks and can be accessed by Playwright. We end up with 4 websites that with at least 8 tasks to run online evaluations (Zheng et al., 2024; Yoran et al., 2024; He et al., 2024) and include 57 tasks, listed in Appendix § D. To ensure evaluation validity, we manually evaluate the success of agent trajectories. Specifically, we verify whether the actions fulfill all requirements posed by the task and whether retrieved information matches the target for information-seeking tasks.

3.3 Agent Implementation

By default, we use GPT-4o[2] with a temperature of 0.3. Following the default WebArena evaluation settings, we set the maximum number of steps per iteration to 10.

Baseline Agent. We implement a web agent with only browsing actions based on CodeAct (Wang et al., 2024b), which leverages an LLM to generate Python code using the Playwright browser automation library to interact with web environments through simple atomic actions such as `click`, `type`, and `scroll`. The observation space consists of webpage screenshots, an accessibility tree, and the execution results from previous actions, such as exceptions.

Agent with Skills. To demonstrate the performance improvements gained from APIs, we extend the action space of the baseline agent by incorporating synthesized skills, enabling the agent to execute API calls. We leverage this agent to evaluate the performance improvements from both APIs synthesized by SKILLWEAVER and human-crafted APIs (Song et al., 2024b) from WebArena's official documentation. Given the extensive number of synthesized APIs, we introduce an API selection module that filters and selects only relevant APIs from the API library. This module also removes APIs that do not satisfy pre-conditions. The prompt and skill serialization format can be found in Appendix § B.8.

Weaker Agent. To evaluate whether the synthesized APIs can plug and play in weaker agents, we implement a weaker variant by replacing GPT-4o with GPT-4o-mini[3] while maintaining the same agent design.

3.4 Exploration

During the exploration stage, we employ the agent described above to practice skills. Each website undergoes an exploration process consisting of 160 iterations with GPT-4o, where an iteration is defined as either attempting a proposed skill or testing an existing skill. Within each iteration, the agent proposes either a procedural or navigational task at the starting webpage. It can also propose an information-seeking task at the ending webpage after completing the procedural or navigational task in the previous iteration. During exploration, the agent can access APIs synthesized in the previous exploration steps after skill selection to compose more complex APIs.

4 Results and Analysis

4.1 Experimental Results

WebArena Results. Our experiments on WebArena demonstrate consistent performance improvements with the integration of synthesized APIs. As illustrated in Table 1, we observe a substantial relative improvement in success rate, 39.8% on average, for the baseline agent with GPT-4o and an even larger improvement of 54.3% with GPT-4o-mini across the evaluated websites.

When compared to AutoEval (Pan et al., 2024), which leverages an LLM-based reward model to guide inference-time exploration, SKILLWEAVER achieves higher average success rates and exhibits better or comparable performance across all domains, with the exception of the Shopping environment. The reason behind this exception is the inherent requirements for more extensive interaction with such websites for partially observable information, like dynamic product search results and product details. In comparison with SteP (Sodhi et al., 2024), which incorporates the external memory of domain-specific human-written workflows, SKILLWEAVER achieves better performance on CMS and Map environments, showing the promising quality of agent synthesized APIs even comparing with manually crafted workflows designed by human experts.

[2]Azure Endpoint: `gpt-4o-2024-08-06`
[3]Azure Endpoint: `gpt-4o-mini-2024-07-18`

Method	Gitlab	Map	Shopping	CMS	Reddit	AVG.
WebArena	15.0	15.6	13.9	10.4	6.6	12.3
AutoEval	25.0	27.5	39.6	20.9	20.8	26.9
*SteP	32.0	30.0	37.0	24.0	59.0	33.0
SKILLWEAVER						
GPT-4o	17.8	27.5	19.8	18.7	37.7	22.6
+ Skills	22.2	33.9	27.2	25.8	50.0	29.8
Δ	↑ 25%	↑ 23%	↑ 38%	↑ 38%	↑ 33%	↑ 32%
GPT-4o-mini	6.1	10.3	11.8	3.3	18.9	9.2
+ Skills	8.9	16.7	17.1	7.7	26.4	14.1
Δ	↑ 46%	↑ 62%	↑ 46%	↑ 133%	↑ 40%	↑ 45%

Table 1: Task success rate on WebArena. The numbers in green represent the relative improvement percentage.

Method	Drug	Flight	Cooking	Car	AVG.
Baseline	65.0	11.7	62.5	11.1	40.2
+ Skills	87.0	29.4	75.0	11.1	56.2
Δ	↑ 34%	↑ 151%	↑ 20%	↑ 0%	↑ 40%

Table 2: Success rate in real-world website evaluation based on Online-Mind2Web.

Live Website Results. As shown in Table 2, integrating synthesized APIs leads to an average relative improvement of 39.8% in success rate across four websites. For the Car website, while the final success rate remained unchanged after incorporating synthesized APIs, we observed that in four out of nine tasks, the synthesized APIs successfully guided the agent to the final state required for task completion. However, the agent still encountered failures during the final steps that demanded strong environment understanding and visual reasoning.

4.2 Analysis

Generalization across Website. Generalization across diverse websites, particularly less common websites, remains a significant challenge. SKILLWEAVER presents a new angle to this challenge by integrating website-specific skills and knowledge through external memory in synthesized skill files. SKILLWEAVER is able to encapsulate skills and knowledge on website environments into code files containing accumulated skills.

To demonstrate the viability of such generalization, we don't include any website specific in the agent and adapt the agent to different websites by purely integrating synthesized API libraries generated during pre-hoc website explorations. As shown in both Table 1 and Table 2, our agent is not only able to achieve stable performance improvements across all 5 WebArena websites but also in the 4 real-world live websites that are more complex and noisy.

Generalization across Agents. A critical feature of the skill repertoire is its ability to generalize across different web agents. Once the exploration process is completed, the synthesized API library can be reused to expand and enhance the action space of other agents capable of API calling. We conduct a comparative experiment by replacing the backbone language model with GPT-4o-mini without any other modification to the agent.

According to the results in Table 1, after switching the language model to GPT-4o-mini, the performance of the baseline agent is substantially weaker than other agents based on GPT-4o. However, upon integration with our synthesized APIs, the agent based on GPT-4o-mini demonstrates remarkable relative success rate improvements ranging from 40% to 133% across five websites. Notably, this agent even achieves a better average success rate compared to the WebArena agent based on GPT-4o, particularly in the Map, Shopping, and

Reddit websites. These results highlight the transferability of our synthesized APIs as a plug-and-play module to enhance agents.

As the base agents become stronger, particularly those capable of more sophisticated inference-time search (*e.g.*, Operator), the potential of SKILLWEAVER can be further unleashed. While current agents can accomplish challenging tasks through trial-and-error (Song et al., 2024a) and search over webpages Koh et al. (2024), certain complex skills remain beyond their capabilities. For example, as illustrated in Appendix E.2.3, the task `request_quote_for_multiple_listings` requires repeated, sequential, successful searches for specific vehicles, followed by separate quote requests. This process demands both long-range planning and back-tracing capabilities, which present significant challenges for existing agents. As agents grow stronger, SKILLWEAVER will enable the synthesis of increasingly sophisticated and complex skills.

Comparison with Human-Crafted APIs. We further conducted a comparison of the performance of agent with synthesized skills with the human-crafted APIs extracted from the official documentation and outsourcing websites (Song et al., 2024a). Based on the level of API support, websites are classified into three categories: low, medium, and high.

As shown in Figure 3, the performance of the synthesized APIs is comparable with human-crafted APIs on websites with low API support, such as Reddit, and those with medium API support, like Shopping. This result suggests that our exploration process is capable of generating APIs with quality that is comparable to or even superior to manually crafted APIs found in official documentation. On websites with high API support, such as GitLab and Maps, the performance of the synthesized APIs is not as good.

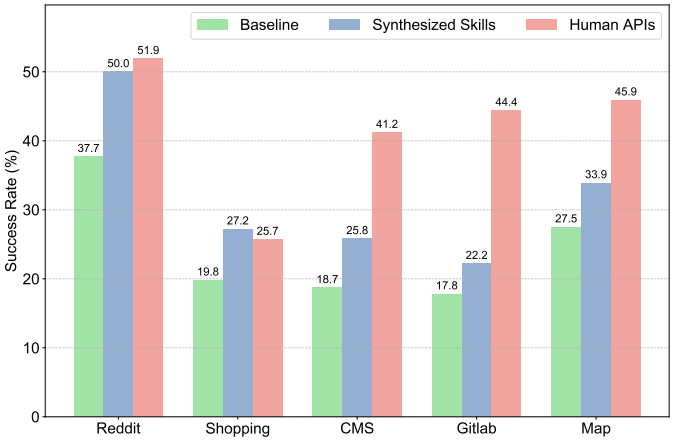

Figure 3: Success rate with synthesized vs. human-crafted APIs.

4.3 Case Studies

Emergence of Compositional APIs. After a certain number of exploration iterations, we observe that the pipeline begins to generate *compositional APIs* that call multiple simpler APIs. An example is shown in § E.3. This API is responsible for applying multiple filters to search results that involves sequentially calling another API to close a webpage overlay, followed by two additional APIs to refine the search. This emergent behavior indicates an increasing level of abstraction and efficiency in SKILLWEAVER 's ability to synthesize and integrate discovered skills.

Limitation in Skill Use. LLMs like GPT-4o are still not robust enough at API calling, and that hurts API-augmented agents like ours, even with human-crafted APIs. This challenge is even greater for weaker LLMs such as GPT-4o-mini, as illustrated in Table 1. We identify two primary categories of failures: (1) failure to identify the appropriate API and (2) generating wrong parameters. For example, SKILLWEAVER synthesized APIs to augment agents to search recipes in Cookpad.com. As demonstrated in Appendix E.2.4, the LLM is not able to identify the `search_recipes_by_cuisine_type('hamburger')` API to finish the task "Save a hamburger recipe". In Appendix E.2.5, the LLM successfully identifies the right API `search_recipes_by_ingredients(page, 'ingredients')` to complete the task "Browse recipes for gluten-free chocolate chip cookies that can be made without nuts". However, it generates the wrong keyword 'chocolate chip, -nuts' instead of 'chocolate chip without nuts', leading to empty search results.

5 Related Work

Web Agents. Automated web agents have recently emerged as a critical research direction with the vision of automating workflows in the internet. Substantial progress in web agents has been achieved by leveraging powerful LLMs alongside diverse prompting strategies to automate the prediction of web page actions (Zheng et al., 2024; Furuta et al., 2024; Zhou et al., 2024a; Yao et al., 2022). Given inputs like HTML content, accessibility trees, and screenshots, LLM-based approaches use in-context examples, skill usage, and the ReAct framework (Yao et al., 2023) to improve action prediction. These models can be further enhanced with action–coordinate pair data for better grounding (Cheng et al., 2024; Gou et al., 2024; You et al., 2024), learning from multiple web agent trajectories to improve multi-step reasoning and planning (Lai et al., 2024; Shaw et al., 2023; Deng et al., 2023), or training the agent to directly output pixel coordinates (Hong et al., 2024). Another line of work uses planning. Tree search agent (Koh et al., 2024) performs a best-first tree search using an LLM as a heuristic, and AgentQ (Putta et al., 2024) and WebPilot (Zhang et al., 2024) employ Monte Carlo Tree Search. The World-model-augmented web agent (Chae et al., 2024) and WebDreamer (Gu et al., 2024b) integrate LLM-based world models to anticipate the outcomes of its actions.

Tool Use and Synthesis. With the great potentials exhibited by tool-augmented LMs (Schick et al., 2023; Wang et al., 2024c), many works explore using LLMs to make tools across math reasoning (Cai et al., 2024; Qian et al., 2023; Yuan et al., 2024a), theorem proving (Wang et al., 2024a), structured data analysis (Lu et al., 2023; Wang et al., 2024d), and digital agent automation (Wang et al., 2023a; Gu et al., 2024a; Song et al., 2024b). However, existing methods rely on a set of existing training examples to determine tool correctness (Yuan et al., 2024a), or user input queries to bootstrap the tool-making system (Wang et al., 2024d). Our work does not necessitate annotated data and enables agents to gather experiences by self-exploring the environment while synthesizing more skills.

Self-Improvement. Many works explore collecting trajectories through experience, rating the success with a reward model, and using the results for policy training (Patel et al., 2024; Huang et al., 2023; Madaan et al., 2023; Wang et al., 2023b; Singh et al., 2024; Chen et al., 2024; Yuan et al., 2024b). These approaches can be broadly categorized into parametric and non-parametric approaches. Parametric training with exploration trajectories has been investigated using various strategies, such as supervised fine-tuning (e.g., WebGUM (Furuta et al., 2024), Patel et al. (2024), and ScribeAgent (Shen et al., 2024)) and reinforcement learning–style optimization (e.g., ETO (Song et al., 2024a) and PAE (Zhou et al., 2024b)). Non-parametric methods typically transform exploration trajectories into reusable APIs or workflows that can be composed into more complex skills, like Voyager (Wang et al., 2023a) an LLM-based lifelong learning agent that continually explores the Minecraft environment. Friday (Wu et al., 2024) also investigates self-improvement on operating systems through exploration with pre-defined learning objectives. For web agents, the collected trajectories can be used as examples for in-context learning (Murty et al., 2024b), provide targets for fine-tuning (Furuta et al., 2024; Patel et al., 2024; Song et al., 2024a), and be distilled into text instructions to reproduce the desired behavior (workflows) (Wang et al., 2024e). Our work focuses on enabling web agents to continuously expand their skill repertoire by autonomously exploring website environments and distilling successful trajectories into reusable skills represented as Python APIs. These APIs are created to enable the agent to operate more efficiently during subsequent exploration and task execution.

6 Conclusion

Web agents aim to automate browsing tasks to enhance human productivity across diverse digital environments. A key challenge lies in adapting to real-world websites characterized by high diversity and complexity. Inspired by the self-improvement mechanism of human, we propose SKILLWEAVER, a skill-centric framework enabling web agents to autonomously self-improve through exploration and API synthesis. Experimental results demonstrate substantial performance improvements, with relative success rate gains of 31.8% on WebArena benchmark and 39.8% on real-world websites. Notably, weaker agents equipped with skills

from more powerful counterparts showed improvements of up to 54.3%, demonstrating effective knowledge distillation through lightweight, transferable modules. Our findings point to a promising future where web agents can learn to improve themselves without parameter tuning by evolving a collective skill base.

Ethics Statement

Generalist web agents hold the potential to automate routine web tasks, enhance user experiences, and promote web accessibility, safety concerns related to their real-world deployment are also critical. These concerns span privacy issues, such as access to users' personal profiles, and sensitive operations, such as financial transactions or application form submissions. During online evaluation, we noticed the possibility of these web agents generating harmful actions on the web, and we manually validated the safety of all the actions before execution. Exploration on live website may lead to potential safety concerns about trigger harmful actions. In case the agent is being used on a live website, we add safety instructions which guardrail the agent away from actions that could potentially have side effects or that would otherwise be disadvantageous to explore, such as creating accounts or interacting with human users. It is critical for further research to thoroughly assess and mitigate the safety risks associated with web agents, ensuring they are safeguarded against producing and executing harmful actions. The code will also be released solely for research purposes, with the goal of making the web more accessible via language technologies under an OPEN-RAIL license. We are strongly against any potentially harmful use of the data or technology by any party.

References

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=qV83K9d5WB.

Hyungjoo Chae, Namyoung Kim, Kai Tzu iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. Web agents with world models: Learning and leveraging environment dynamics in web navigation. *ArXiv*, abs/2410.13232, 2024. URL https://api.semanticscholar.org/CorpusID:273404026.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=O4cHTxW9BS.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing GUI grounding for advanced visual GUI agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 9313–9332. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.505. URL https://doi.org/10.18653/v1/2024.acl-long.505.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samual Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/5950bf290a1570ea401bf98882128160-Abstract-Datasets_and_Benchmarks.html.

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. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 8657–8677. PMLR, 2023. URL https://proceedings.mlr.press/v202/du23f.html.

Hiroki Furuta, Kuang-Huei Lee, Ofir Nachum, Yutaka Matsuo, Aleksandra Faust, Shixiang Shane Gu, and Izzeddin Gur. Multimodal web navigation with instruction-finetuned foundation models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=efFmBWioSc.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. Navigating the digital world as humans do: Universal visual grounding for gui agents. *arXiv preprint arXiv:2410.05243*, 2024.

Yu Gu, Yiheng Shu, Hao Yu, Xiao Liu, Yuxiao Dong, Jie Tang, Jayanth Srinivasa, Hugo Latapie, and Yu Su. Middleware for llms: Tools are instrumental for language agents in complex environments. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pp. 7646–7663. Association for Computational Linguistics, 2024a. URL https://aclanthology.org/2024.emnlp-main.436.

Yu Gu, Boyuan Zheng, Boyu Gou, Kai Zhang, Cheng Chang, Sanjari Srivastava, Yanan Xie, Peng Qi, Huan Sun, and Yu Su. Is your llm secretly a world model of the internet? model-based planning for web agents, 2024b. URL https://arxiv.org/abs/2411.06559.

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. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 6864–6890. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.371. URL https://doi.org/10.18653/v1/2024.acl-long.371.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and Jie Tang. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14281–14290, 2024.

Jiaxin Huang, Shixiang Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pp. 1051–1068. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EMNLP-MAIN.67. URL https://doi.org/10.18653/v1/2023.emnlp-main.67.

Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. *CoRR*, abs/2407.01476, 2024. doi: 10.48550/ARXIV.2407.01476. URL https://doi.org/10.48550/arXiv.2407.01476.

Hanyu Lai, Xiao Liu, Iat Long Iong, Shuntian Yao, Yuxuan Chen, Pengbo Shen, Hao Yu, Hanchen Zhang, Xiaohan Zhang, Yuxiao Dong, et al. Autowebglm: A large language model-based web navigating agent. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5295–5306, 2024.

Wei Li, William E. Bishop, Alice Li, Christopher Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the effects of data scale on UI control agents. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances*

in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/a79f3ef3b445fd4659f44648f7ea8ffd-Abstract-Datasets_and_Benchmarks_Track.html.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/871ed095b734818cfba48db6aeb25a62-Abstract-Conference.html.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Shikhar Murty, Dzmitry Bahdanau, and Christopher D. Manning. Nnetscape navigator: Complex demonstrations for web agents without a demonstrator. *CoRR*, abs/2410.02907, 2024a. doi: 10.48550/ARXIV.2410.02907. URL https://doi.org/10.48550/arXiv.2410.02907.

Shikhar Murty, Christopher D. Manning, Peter Shaw, Mandar Joshi, and Kenton Lee. BAGEL: bootstrapping agents by guiding exploration with language. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024b. URL https://openreview.net/forum?id=VsvfSMI5bs.

Vardaan Pahuja, Yadong Lu, Corby Rosset, Boyu Gou, Arindam Mitra, Spencer Whitehead, Yu Su, and Ahmed Awadallah. Explorer: Scaling exploration-driven web trajectory synthesis for multimodal web agents. 2025. URL https://api.semanticscholar.org/CorpusID:276408442.

Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. Autonomous evaluation and refinement of digital agents. *CoRR*, abs/2404.06474, 2024. doi: 10.48550/ARXIV.2404.06474. URL https://doi.org/10.48550/arXiv.2404.06474.

Ajay Patel, Markus Hofmarcher, Claudiu Leoveanu-Condrei, Marius-Constantin Dinu, Chris Callison-Burch, and Sepp Hochreiter. Large language models can self-improve at web agent tasks. *CoRR*, abs/2405.20309, 2024. doi: 10.48550/ARXIV.2405.20309. URL https://doi.org/10.48550/arXiv.2405.20309.

Pranav Putta, Edmund Mills, Naman Garg, Sumeet Motwani, Chelsea Finn, Divyansh Garg, and Rafael Rafailov. Agent Q: advanced reasoning and learning for autonomous AI agents. *CoRR*, abs/2408.07199, 2024. doi: 10.48550/ARXIV.2408.07199. URL https://doi.org/10.48550/arXiv.2408.07199.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://openreview.net/forum?id=aCHq10rQiH.

Gabriel Sarch, Lawrence Jang, Michael J. Tarr, William W. Cohen, Kenneth Marino, and Katerina Fragkiadaki. Ical: Continual learning of multimodal agents by transforming trajectories into actionable insights. *ArXiv*, abs/2406.14596, 2024. URL https://api.semanticscholar.org/CorpusID:274466462.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

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. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/6c52a8a4fadc9129c6e1d1745f2dfd0f-Abstract-Conference.html.

Junhong Shen, Atishay Jain, Zedian Xiao, Ishan Amlekar, Mouad Hadji, Aaron Podolny, and Ameet Talwalkar. Scribeagent: Towards specialized web agents using production-scale workflow data. *ArXiv*, abs/2411.15004, 2024. URL https://api.semanticscholar.org/CorpusID:274192657.

Avi Singh, John D. Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J. Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron T. Parisi, Abhishek Kumar, Alexander A. Alemi, Alex Rizkowsky, Azade Nova, Ben Adlam, Bernd Bohnet, Gamaleldin Fathy Elsayed, Hanie Sedghi, Igor Mordatch, Isabelle Simpson, Izzeddin Gur, Jasper Snoek, Jeffrey Pennington, Jiri Hron, Kathleen Kenealy, Kevin Swersky, Kshiteej Mahajan, Laura Culp, Lechao Xiao, Maxwell L. Bileschi, Noah Constant, Roman Novak, Rosanne Liu, Tris Warkentin, Yundi Qian, Yamini Bansal, Ethan Dyer, Behnam Neyshabur, Jascha Sohl-Dickstein, and Noah Fiedel. Beyond human data: Scaling self-training for problem-solving with language models. *Trans. Mach. Learn. Res.*, 2024, 2024. URL https://openreview.net/forum?id=lNAyUngGFK.

Paloma Sodhi, S. R. K. Branavan, Yoav Artzi, and Ryan McDonald. Step: Stacked llm policies for web actions, 2024. URL https://arxiv.org/abs/2310.03720.

Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. Trial and error: Exploration-based trajectory optimization for LLM agents. *CoRR*, abs/2403.02502, 2024a. doi: 10.48550/ARXIV.2403.02502. URL https://doi.org/10.48550/arXiv.2403.02502.

Yueqi Song, Frank F. Xu, Shuyan Zhou, and Graham Neubig. Beyond browsing: Api-based web agents. *ArXiv*, abs/2410.16464, 2024b. URL https://api.semanticscholar.org/CorpusID:273507298.

Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan Ö. Arik. Learn-by-interact: A data-centric framework for self-adaptive agents in realistic environments. *CoRR*, abs/2501.10893, 2025. doi: 10.48550/ARXIV.2501.10893. URL https://doi.org/10.48550/arXiv.2501.10893.

Fahim Tajwar, Yiding Jiang, Abitha Thankaraj, Sumaita Sadia Rahman, J. Zico Kolter, Jeff Schneider, and Ruslan Salakhutdinov. Training a generally curious agent. *CoRR*, abs/2502.17543, 2025. doi: 10.48550/ARXIV.2502.17543. URL https://doi.org/10.48550/arXiv.2502.17543.

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. *Trans. Mach. Learn. Res.*, 2024, 2023a. URL https://api.semanticscholar.org/CorpusID:258887849.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. LEGO-prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=3f5PALef5B.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024b. URL https://openreview.net/forum?id=jJ9BoXAfFa.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pp. 13484–13508. Association for Computational Linguistics, 2023b. doi: 10.18653/V1/2023.ACL-LONG.754. URL https://doi.org/10.18653/v1/2023.acl-long.754.

Zhiruo Wang, Zhoujun Cheng, Hao Zhu, Daniel Fried, and Graham Neubig. What are tools anyway? a survey from the language model perspective. In *First Conference on Language Modeling*, 2024c. URL https://openreview.net/forum?id=Xh1B90iBSR.

Zhiruo Wang, Graham Neubig, and Daniel Fried. TroVE: Inducing verifiable and efficient toolboxes for solving programmatic tasks. In *Forty-first International Conference on Machine Learning*, 2024d. URL https://openreview.net/forum?id=DCNCwaMJjI.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024e.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement. *CoRR*, abs/2402.07456, 2024. doi: 10.48550/ARXIV.2402.07456. URL https://doi.org/10.48550/arXiv.2402.07456.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *CoRR*, abs/2404.07972, 2024. doi: 10.48550/ARXIV.2404.07972. URL https://doi.org/10.48550/arXiv.2404.07972.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? assessing the current state of web agents. *OSU NLP Blog*, Mar 2025. URL https://tinyurl.com/online-mind2web-blog.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/82ad13ec01f9fe44c01cb91814fd7b8c-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. Assistantbench: Can web agents solve realistic and time-consuming tasks? In *Conference on Empirical Methods in Natural Language Processing*, 2024. URL https://api.semanticscholar.org/CorpusID:271328691.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. Ferret-ui: Grounded mobile UI understanding with multimodal llms. In Ales Leonardis, Elisa Ricci, Stefan Roth, Olga Russakovsky, Torsten Sattler, and Gül Varol (eds.), *Computer Vision - ECCV 2024 - 18th European Conference, Milan, Italy, September 29-October 4, 2024, Proceedings, Part LXIV*, volume 15122 of *Lecture Notes in Computer Science*, pp. 240–255. Springer, 2024. doi: 10.1007/978-3-031-73039-9_14. URL https://doi.org/10.1007/978-3-031-73039-9_14.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Fung, Hao Peng, and Heng Ji. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=G0vdDSt9XM.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024b. URL https://openreview.net/forum?id=0NphYCmgua.

Yao Zhang, Zijian Ma, Yunpu Ma, Zhen Han, Yu Wu, and Volker Tresp. Webpilot: A versatile and autonomous multi-agent system for web task execution with strategic exploration. *CoRR*, abs/2408.15978, 2024. doi: 10.48550/ARXIV.2408.15978. URL https://doi.org/10.48550/arXiv.2408.15978.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. Gpt-4v(ision) is a generalist web agent, if grounded. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=piecKJ2DlB.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024a. URL https://openreview.net/forum?id=oKn9c6ytLx.

Yifei Zhou, Qianlan Yang, Kaixiang Lin, Min Bai, Xiong Zhou, Yu-Xiong Wang, Sergey Levine, and Li Erran Li. Proposer-agent-evaluator(pae): Autonomous skill discovery for foundation model internet agents. *CoRR*, abs/2412.13194, 2024b. doi: 10.48550/ARXIV.2412.13194. URL https://doi.org/10.48550/arXiv.2412.13194.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, et al. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*, 2024.

Table of Content:

A Agent Definitions

A.1 Code Generation Agent Action Space

We represent the action space as the space of Python programs. When using a knowledge base, all function declarations are prepended to the prompt, surrounded by some descriptive text. Then, the agent is prompted to generate a function called `act`, which uses a Python object for the current tab as an input argument called `page`. Then, the agent can call Playwright APIs on the page, using Playwright's accessibility tree interaction APIs. These APIs allow the agent to interact with an element given its accessible role, and accessible name. Alternatively to writing Python code, the agent can specify a "terminate" action, with a result from the trajectory.

A.1.1 Validation Checks for Code Generation Agent

We perform several validation checks on all results from the model. By using these checks, we can forecast that code will not succeed in execution (or not be useful for a tool). This in turn increases the chance that functional tools are generated and reduces the frequency of partially-executed actions (which produce Exceptions, which must be placed in the prompt). Therefore, another effect is that this guards the context from messy information.

1. **Syntax**: `ast.parse` must execute without error.
2. **Format**: Generated tools must be asynchronous functions with `page` as the first argument.
3. **Static type checks**: We use PyRight, a static type checker from Microsoft, and Python's static type annotations (`page:Page`) to verify that the Playwright API is used correctly.
4. **Avoiding error-prone Playwright APIs**: The bot must use ARIA information to select elements, rather than CSS-style locators. This is because the LLM is presented website content in an accessibility tree format, and therefore any CSS-style selector must be a hallucination. Additionally, the bot must create a locator using the `page.locator(...)` syntax for interacting with elements, instead of the deprecated `page.click(...)`, `page.fill(...)`, and `page.type(...)` APIs.
5. **Proactive error-handling**: `try` statements are forbidden.
6. **Avoiding infinite loops**: `while` loops are forbidden.

Sanity checks: After the agent generates an action, we sanity check it. This is done by validating the syntax using Python's built-in `ast` module; performing a type check using `Pyright`, a static type checker, with Playwright APIs; and avoiding common hallucinations, like Playwright APIs that use CSS selectors instead of accessibility tree functions. Whenever a check fails to pass, we generate an error message, and append the failing code and error message along with new instructions to the prompt. We attempt this three times before terminating the step with a no-op.

A.2 Function-Calling Agent Action Space

We represent the action space as the following set of functions. These are passed as parameters to the OpenAI API, which uses structured outputs to ensure that parameters are correct. We extend SeeAct by using an accessibility tree hierarchy instead of a flat multiple-choice question answering prompt. For each interactable element in the hierarchy, we annotate it with multiple choice symbols ("A", "B", "C", etc.), which are used to point to the target element of each interaction. We elaborate on the functions provided in the Appendix.

1. `click(element_choice_name)`: Click on an element on the page.
2. `type(element_choice_name, text)`: Type text into an element on the page.
3. `hover(element_choice_name)`: Hover over an element on the page.

4. `select_option(element_choice_name, option)`: Select an option from a dropdown on the page.
5. `go_back()` : Go back to the previous page.
6. `go_forward()` : Go to the next page.
7. `scroll_up()` : Scroll up half a screen.
8. `scroll_down()` : Scroll down half a screen.
9. `terminate(result, success)` : Terminate your attempt, providing a result or failure reason.

B Prompts

B.1 Skill Proposal

In case the agent is being used on a live website, we add safety instructions which bias the agent away from actions that could potentially have side effects or that would otherwise be disadvantageous to explore, such as creating accounts or interacting with human users.

Proposing Tasks to Explore

System: You propose tasks that would make good 'tools' for external users of a website. **User**:

```
You are a 'web agent' who is learning how to use a website. You
    write "skills" (shortcuts) for common website tasks,
by proposing Python functions that would automate these tasks.

You have already proposed the following skills:
<proposed>
{procedural_knowledge}
</proposed>

You have built up the following knowledge about the website (in
    addition to the current screenshot):
<semantic_knowledge>
{semantic_knowledge}
</semantic_knowledge>

Now please come up with something new to learn how to do on this
    website. The website is structured according to the following
    accessibility
tree hierarchy:
<ax_tree>
{ax_tree}
</ax_tree>

Do not interact with the Advanced Reporting tab if you are using
    Magento.
Do not interact with login/logout/user accounts on any site.
If you're on OpenStreetMap, don't interact with community features.

Write a list of useful skills/shortcuts that you would want to have
     built into a website as Python functions. Write the name
in natural language format. Do not use "*_id" as a parameter in
    your skill. Again, your goal is to generate functions
that would be useful "shortcuts" for users of the website, so you
    should prioritize generating skills that compress
a couple interactions into a single function call. Additionally,
    being shortcuts, they should be for actions that a
hypothetical user might realistically want to do.

Then, estimate:
```

```
(1) how useful they are (5 being difficult and frequency, 1 being
    trivial or uncommon),
(2) the expected number of clicking/typing actions required to
    complete the skill. (calculate this by writing the list of steps
     and counting AFTERWARDS)

Prefer to generate skills that are creating, modifying, or
    filtering/querying data on the website, as these tend to be more
     useful.
Do not generate skills simply to perform single clicks.

{safety_instructions}

Then, calculate the sum of these ratings for each skill.  Finally,
    select the skill with the highest rating.
Write your ratings in `step_by_step_reasoning`. Then, write your
    skill choice in `proposed_skill`.
```

B.2 Code Generation Agent Context Representation

B.2.1 States as Accessibility Trees

We represent the state as a tuple containing an screenshot of the page, a title, a URL, and an accessibility tree hierarchy. These are serialized by concatenating clean string representations of each of these properties. We represent each node in the accessibility hierarchy as a line containing (1) indentation representing the depth in the tree, (2) the accessible role, (3) the accessible name of the element, in quotation marks, and finally (4) any other accessibility tree information (represented as `aria-` attributes in the DOM.)

B.2.2 Prompts

Substrings in square brackets (`[]`) are omitted if not applicable.

Code Agent Context Representation

```
<state 0>
URL: {relative_url}

{ state_accessibility_tree_string }
</state>

<reasoning>
{step_by_step_reasoning}
</reasoning>

<code>
{action_source}
</code>

[<stdout>
{stdout}
</stdout>]

[<result>
{return_value}
</result>]

[<warnings>
{warnings}
```

</warnings>]

[<recovery>
{recovered_exception}
</recovery>, ...]

[<exception>
{unrecoverable_exception}
</exception>]

<state 1>
URL: {relative_url}

{ state_accessibility_tree_string }
</state>

...

<state N>
URL: {relative_url}

{ state_accessibility_tree_string }
</state>

<terminate_with_result>
{result}
</terminate_with_result>

B.3 Code Generation Agent Error Detection

This is the prompt template used if we detect any errors in the agent's code. We use the following error messages:

1. A function named `act` not provided: `Function name must be 'act'`
2. The function `act` does not contain a single argument, `page`: `Function must take exactly one argument: 'page'.`
3. The function `function` is disabled: `Function 'fnname' is disabled. Please use another function.`
4. Incorrect or buggy Playwright functions: `Please use the 'page.get_by_...().f()' functions instead of the 'page.f(selector)' functions.` where $f \in \{click, fill, type\}$.
5. Use of CSS selectors instead of accessibility tree selectors: `Please use Accessibility Tree-centric selectors, like 'page.get_by_role()', '.nth()', instead of the CSS-style selectors like '.locator()' or '.query_selector()'.`
6. Blank response: `You did not provide any Python code, but you also did not provide a result for 'terminate_with_result'. Please provide one or the other.`
7. Type errors: `Type Error: {error}`

B.4 Action Synthesis

Code Agent Action Generation

You generate Playwright code to interact with websites. Words of wisdom:

- If you want to click a generic button (e.g., that belongs to an element), use the full `.get_by_role()` path to the element (e.g., `.get_by_role("group", name="Test Item").get_by_role("button", name="Go")` instead of `.get_by_role("button", name="Go")`, as this is ambiguous).
- Write exceptionally *correct* Python code.
- You love to take advantage of functions in the `knowledge_base` whenever possible. You use them via Python function calls. It is required to use the knowledge base function corresponding to an action if it exists.
- Use relative `goto` when you can.

You are currently on a webpage titled {`title_repr`}, with the URL {`url_repr`}.
Consider the following Accessibility Tree. The accessibility tree is formatted like this:

```
[role] ["name"] [properties] {{
    [child1]
}};
[role] ["name"] [properties]; // no children
```

Here is an example:

```
<example>
article "Willy Wonka's Chocolate Factory Opens Its Gates" {{
    link "Share";
}};

article "Hogwarts Begins Accepting Applicants for Class of 2029" {{
    link "Share";
}};
</example>
```

To select the first link, you can do (because name = case-insensitive substring match unless `exact=True` is supplied):

```
first_link = page.get_by_role(" article ", name="Willy Wonka").get_by_role("link", name="
    Share")
```

Here is another example:

```
<example>
article {{
    header "How can I install CUDA on Ubuntu 22?";
    link "Share";
}};

article {{
    header "How do I install Docker?";
    link "Share";
}};
</example>
```

To select the first link, you can do:

```
page.get_by_role(" article "). filter (has=page.get_by_role("header", "How can I install
    CUDA")).get_by_role("link", name="Share")
```

Here, the `.filter()` is necessary to ensure that we select the correct `article`. This is especially important in cases where the parent element doesn't include relevant criteria for identifying a child element.
Tips about this tree:

- If you see a node as a child of an `iframe`, you must use `page.frame(name=...)`, and *then* access the node (via `.get_by_role()` or similar).

- Some elements will require interaction via `.select_option()`. They will be labeled as such. This is because they are HTML `<select>` elements, and thus cannot be interacted with via clicks.

You also have this library of Python functions available to you. Think carefully about whether you think these can be used in your code. If you conclude that you can use a function, then simply call it. These functions are all available as global variables.

```
<functions>
{functions}
</functions>
```

Additionally, you have collected the following knowledge about the website:

```
<semantic_knowledge>
{semantic_knowledge}
</semantic_knowledge>
```

Please think step by step and determine if further action needs to be taken. `{step_specific_instructions}`
If further action needs to be taken, provide a code snippet to interact with the page. Use Playwright and complete the following Python function declaration:

```
async def act(page):
    ...
```

If you have no more actions to perform, leave the `python_code` field blank and set the `terminate_with_result` field of your response to a non-empty string.
Your previous actions are:

```
<previous_actions>
{previous_actions}
</previous_actions>
```

Here is the current website state:

```
<webpage_accessibility_tree>
{ax_tree}
</webpage_accessibility_tree>
```

Your previous actions have already executed. So your `act()` function should only represent **NEW** actions you want to take from here.
IMPORTANT: DO NOT CHANGE LANGUAGE SETTINGS! Keep everything in English, even if you are in exploration mode.

Additionally, if you click a link that you expect will take you away from the current page, make that the last action in `act()`. Don't write selectors that follow page navigations because you can't predict what they will be.

Your task is `{task}`.

Make sure to continue from the current state of the UI—not necessarily the initial state. For example, if a dropdown is open, you do not need to open it again, and things like that. If it appears that the website is in a loading state, you may call `await asyncio.sleep(5)` to wait for the page to load.

```
{additional_instructions}
```

B.5 Success Checking

Responses to this were generated using Structured Outputs, as a dictionary with the keys `step_by_step_reasoning`, a string, and `success`, a Boolean.

Success Verification

Please observe the following action log of someone using a computer. They were trying to do the following task: `task`. Given this log and the screenshot of their screen at the end of their attempt, please conclude whether they were successful.
Trajectory: `{trajectory_string}`

B.6 Persisting Attempt to Knowledge Base

Persisting Attempt to Knowledge Base

You are learning how to use a website. In addition to maintaining a semantic base of knowledge, you are also building a procedural knowledge base. You should update your knowledge base with Python functions to automate what you are able to do successfully. These Python functions are written using the Playwright API. Write 'skills ', which are Python code snippets representing logical procedures to perform the task you just completed. Try to make this logical procedure represent the general case of your task, rather than a specific case. Make sure to carefully document the behavior of these skills, especially any behavior that you observe that is unexpected and useful to know for future users. Any examples of what happened after you called the skill will be useful. If you got feedback on whether your results were successful or unsuccessful, make sure to document this feedback as well, and any suggestions to improve performance.

You may be given either a single task attempt and success evaluation, or multiple task attempts and aggregate results. If a task attempt was successful, you may create a skill to simplify the task. If a task attempt uses a skill, but is ultimately unsuccessful, then you may want to improve the documentation for that skill's Python function, to ensure that the skill can be used correctly in the future.

If you used an existing function and the result was unexpected, make sure to update the documentation for that function to ensure that it can be used correctly in the future. Unexpected results should be documented if they occur. You may also want to provide examples of successful usage too.

Reasoning Template
- Make sure that you take a step back and think about the general procedure you used to complete the task.
- Make sure that you explicitly describe the changes that should be made to the function documentation.

Overall Format
- You will be given either an empty knowledge base, or a knowledge base with existing function declarations.
- In your response, write declarations for new functions or update existing functions.
- If you want to update an existing function, then you must have the function name of the new implementation be an exact match
- Do not write "test_" or "explore_" functions, nor functions that "verify" that a previous skill was completed successfully.

Function Declaration
- You should write a detailed docstring for the function (with triple quotes), describing what it does and how to use it.
  - Make sure that any unexpected behavior is documented.
  - Make sure that the observed behavior and mechanics are carefully documented.
- Make sure your function begins with a `page.goto("/...")` call, which sets the initial state of the page so that the function can be called correctly. Use a relative URL.
- In your docstring, you must include a "usage log" which describes the different times you've used the function and what happened.
- Use `page` as the first argument of your function.
- Do not use `dict` as a type for any parameter.
- Avoid using `*_id` or `*_url` parameters, because these are not human−readable. For example:
  - `item_name` is preferred over `item_id`
  - `post_url` is preferred over `post_url`
  - Exception to this rule: If one of the input fields on the page requires you to input a URL.
  - However, you should not do this if you are just going to `page.goto(item_url)` in your code.

– We will check your code for such parameters!
– Make sure your code is correct, do not use any over complicated Python features (e.g. nested functions, and such).
– Make sure your function is async because you use `await`. Your top level function should be the asynchronous one. Do not use any nested functions!!!
– Do not use a global try–catch statement, if the only thing it does is print or reraise the error. Only catch exceptions that you can truly recover from.
– Do not ``overfit'' your function name to a specific set of task parameters. Instead, try to generalize your parameters.

Selectors
– Note that by default, most string matching in the Playwright API is case–insensitive and only searches for substring matches.
If you want an exact match, you can pass the `exact=True` flag.
– Do not overfit to selectors that include numbers in them (e.g. number of notifications, etc. should be replaced by a regex)
In this case, put `import re` *inside* the method body.

For example, `page.get_by_role(name="Hello")` will match Hello and Hello world .
As another example, `page.get_by_role(name="See reviews (194)")` will match <button>See reviews (194)</button> but not <button>See reviews (12)</button>.
If you instead do `page.get_by_role(name=re.compile(r"See reviews (\d+)"))`, it will match both of them.

Error Handling
– When you encounter exceptions, do not just "print" the error – you should either recover from the exception or reraise it.

Misc
– Do not include actions to close cookie banners, because these will not be necessary during future function calls.

If it appears that the task was not completed successfully, don't write a function. You don't want to assume what a good function will look like if you were unsuccessful.

B.7 Scraping Prompts

Determining Whether Scraping is Needed

The programmer is to write information seeking APIs for webpage.

Please help evaluating whether we need information seeking APIs for the current webpage.

For the webpage, I will provide:
1. Screenshot of the webpage interface
2. A summary of the page content and functionality

Please analyze whether information seeking APIs should be implemented for this page.

Consider:
– Whether the webpage is important for information seeking and the user's needs.
– Whether user need to get data from the webpage.
– Data retrieval needs.

Here are some examples of information seeking tasks:
– get all orders in all pages
– get all users
– retrieve all products

- obtain all reviews
- get all repositories of user

The website content summary: {WEBSITE_CONTENT_SUMMARY}

Response format:
- If APIs are needed: Return 'Yes'. List each required API with its purpose.
- If APIs are not needed: Return 'No'. Brief explanation why

Generating Scraping API

You are a professional website API programmer.
I need you to create runnable Playwright-based APIs based on the following requirements:
I will give you a task description, and you will need to create a Playwright script that accomplishes the task.
The script should be written in PythonThe script should be asynchronous and should be written in a way that is easy to read and understand. The script should be runnable and should not contain any syntax errors.
Please just provide the code in a code block. Do not include any additional text or comments.

Task Description: {TASK_DESCRIPTION}.

Example of desired output format:
```python
async def navigate_to_set_status(page: Page):
    await page.click('[data-qa-selector="user_menu"]')
    await page.click('button:has-text("Set status")')
```

1. The input parameter for the API must include `page`, which is a Playwright `Page` object.
   1. You are already on the correct page, do not need to navigate to another page.
   2. Do not include the browser object.
   3. Do not define the browser in the API code.

2. Please just provide the code in a code block. Do not include any additional text or comments. Do not include Usage. Just return the API code that should be inside the function.

3. Code:
   - Always explicitly handle async/await chaining. When calling async methods that return objects with other async methods. Ensure each async operation in the chain is properly awaited. Use parentheses to clearly show the await order when chaining
   - Common patterns to watch for:
     - WRONG: result = await obj.async_method().another_async_method()
     - CORRECT: temp = await obj.async_method() result = await temp.another_async_method() OR: result = await (await obj.async_method()).another_async_method()
   - For browser automation libraries (like Playwright/Puppeteer):
     - Element selection methods (query_selector, query_selector_all) are async
     - Element properties/methods (inner_text, text_content, click) are often async
     - Always await both the selector and the property access

4. If the task is about information seeking, please make sure the information is as comprehensive as possible.

5. Please review your generated code specifically checking for:
   – Missing await statements
   – Proper async method chaining
   – Correct handling of async property access

6. Code requirements:
   1. Do not include ```await page.wait_for_selector(' selector ') ``` in the code. It always BUG.
   2. Make sure the element selector is correct and precise.
   3. If the page already contains information regarding the task, do not use page.goto() to navigate to the page.
   4. If you need to navigate to a page, use page.goto() with the relative URL.
   5. Please only return fixed API code. Do not include any other code like main().

7. HTML Content (truncated if too long):
{HTML_CONTENT}

Verifying Scraping Result

You are a verification system that checks if the code execution result matches the task requirements.

Task Description:
{TASK_DESCRIPTION}

Code Execution Result:
{RESULT}

Please analyze whether the result matches the task requirements. Think about:
1. Does the result contain required information based on the task description?
2. Does the content make sense given the HTML context?

Compare the extracted data with the content visible in the HTML to ensure accuracy and completeness.

Answer "is_correct" if the result meets all requirements.
If not, provide specific suggestions for improvement.

Current HTML Content (for reference):
{HTML_CONTENT}

Fixing Code for Incorrect Scraping Result

You are a Python debugging expert specializing in web automation with Playwright.
Given the following code, error, and HTML context, analyze the issue and provide a fixed version.

Instructions
1. Analyze the HTML structure to understand the page elements and locate what the task requires.
2. Identify the cause of the error in the code
3. Consider common Playwright issues like:
   – Selector timing issues
4. Provide a complete fixed version of the code
5. The HTML content provided is a truncated version of the webpage structure, because of constraints on the context window size.

6. The input parameter for the API must include `page`, which is a Playwright `Page` object.
   1. Do not include the browser object.
   2. Do not define the browser in the API code.
7. Do not need goto() method because the page already contains the necessary information.
8. Do not include ```await page.wait_for_selector('selector')``` in the code. It always BUG.

Please return ONLY the fixed code without any explanation or markdown formatting within the code block.
The code should be a complete, runnable solution that includes all necessary imports.

Please only return fixed API code. Do not include any other code like main().

Task Description
{TASK_DESCRIPTION}

Original Code
```python
{CODE}
```

Error Information
```
{ERROR_INFO}
```

Current Webpage HTML Structure
```html
{HTML_CONTENT}
```

B.8 Skill Retrieval

Retrieving Skills from Knowledge Base

You are provided a list of Python functions representing action shortcuts that can be taken on a website
(in addition to the basic actions like click, type, hover, select_option, etc.)
You identify which functions could be useful for completing a given task.
You do this by breaking the task down into steps and seeing if any functions may be useful.

You are given the following list of functions/shortcuts:
<functions>
{function_space}
</functions>

Your task is {repr_task}.

For each of the listed functions, please determine (explicitly) whether they are useful for the task.
Then, provide a list of the function names that may be useful to the agent.

The agent is then prompted to generate a Structured Outputs dictionary with the keys `step_by_step_reasoning` (string) and `function_names` (list of strings). These functions are then injected into the prompt, formatted as the following:

Format for Functions Placed in Prompt

```
# Skill : {signature}
{docstring}
```

Where `signature` is the Python code signature of the function (extracted by taking the substring of the function's source up until the first close-parentheses), and `docstring` is the (un-indented) docstring from the function's body, as parsed by Python's `ast` module.

C WebArena Benchmark

WebArena benchmark include the following tasks:

- Gitlab: 180 tasks
- Map: 109 tasks
- Shopping: 187 tasks
- CMS: 182 tasks
- Reddit: 106 tasks

D Real-World Website Tasks

Drugs (23 tasks)

- Show me the most helpful review for clonazepam
- Identify the pill with imprint M366
- Find the page with information about ibuprofen interactions
- Show me the page with information about Adderall side effects
- Find the Alcohol/Food Interactions of the drug Allegra
- Find information regarding Hypersomnia
- Find drug Paracetamol and its side effects
- Identify a pill with pink color and oval shape with 894 5 number on it
- Print the Tylenol Side Effects page
- Show me the latest FDA alerts
- Find the risk Summary of Metformin prescribed for a Pregnant woman
- Find available medical apps
- Find the alternative to the prescription drug Valtrex and compare prices at different pharmacies
- Browse the natural products database
- Find the Drug Interaction Report for viagra and alcohol
- Show side effects of Tamiflu
- Browse Humira dosage information
- Show the list of Medications for Anxiety
- Check drug interaction for melatonin and Folate Forte
- Check the interaction between Novolin N and Novolin R
- Find the interactions between Eulexin and hepatic dysfunction
- Find the side effects of taking Montelukast
- Display the search results for pill 123456, white and round

Cookpad (8 tasks)

- Find a recipe for vegetable soup and then follow the author of the recipe
- Find a recipe for pad thai and print it
- Print a recipe containing shrimp
- Find a recipe for Fried Bombay Duck and Save it
- Browse recipes for gluten-free chocolate chip cookies that can be made without nuts
- Save a hamburger recipe
- Show me recipes for pancakes with wheat and without beetroot
- Find a recipe that includes eggplant and mushrooms

Flights (17 tasks)

- Check the status of flight 6944 on April 6
- Search for the flight status for flight 12345678 leaving on April 7
- Find the cheapest one way flight from Dallas to Houston on 24 May
- Find the lowest fare from JFK, NY to Chicago O'Hare and nearby airports for 1 adult on April 22, one-way

- Show route map and flight cost from Los Angeles to Miami on 12 April
- Check the status of flights from the Los Angeles area to the Boston area tomorrow
- Search for flights from New York City to Chicago and filter the results to show only non-stop flights
- Find flights from Seattle to New York on June 5th and only show those that can be purchased with miles
- Browse nonstop flights from Denver to Phoenix from Jun 1st to July 15th
- Find deals for Las Vegas from New York with the budget of $1300 for premium economy
- Book a first-class flight from Seattle to Portland leaving April 13 and returning on April 14
- Book a one-way, fastest, and most flexible direct flight ticket for two from Atlanta to Orlando on March 29 evening
- Book the cheapest economy flight between Miami and Houston for two adults on May 4 with a return option on May 8
- Find a round trip from Phoenix to Miami with a maximum budget of $2000
- Search for a flight from San Francisco to Los Angeles for 2 adults leaving Apr 26 and returning May 2 with the promo code 10000001
- Find one-way flights for one person from Sacramento to Houston IAH on June 2, 2023, that can be booked using miles
- check cheap flights from NYC to Chicago on the 23rd of April for students over 18 years

Cars (9 tasks)

- Find a highest rated dealer for Cadillac with rating above 4 star within 20 miles of zip 60606
- Compare the Acura CL 2003 with the ILX 2022
- Find the seller info and seller's notes about used car model 2011 BMW 135 with a max price of $30000
- Find electric cars with a maximum price of $50,000 within 50 miles of 10001
- Find a cheapest hatchback car listing in Madison which has red interiors with heated seat option and premium sound system
- Find a used cheapest 8 cylinder bmw made between 2005-2015 and priced from $25,000 to $50,000 with mileage less then 50,000 miles or less
- Calculate the estimated car loan payment amount for an average credit-rated person for a $15,000 car with a down payment of $2000 and loan tenure of 48 months in zip 65215 and shop for the lowest priced car
- Find me the cheapest Dodge Ram 1500 within 50 miles of 21122
- Compare Audi A7 with Audi A6 both made in 2023 and hide similarities

E Example APIs

E.1 Success Cases

We find that APIs are able to successfully automate multiple atomic actions and represent them as a single action with richer input parameters. Here are some positive examples.

E.1.1 Correctly-Implemented Skills

Here is an skill that was implemented on the shopping website. It automatically performs the checkout process with the items currently in the cart.

express_checkout

```
async def express_checkout(page):
    """
    Perform an express checkout for the items currently in the
    cart.

    Args:
        page: The Playwright page object to perform actions on.

    Usage Log:
    - Successfully completed express checkout, resulting in an
    order confirmation page with order number 000000191.
    - Initial attempts failed due to a timeout error when clicking
     'Proceed to Checkout'. Resolved by ensuring items were in the
     cart.

    Note:
    - Ensure that the cart is pre-filled with the desired items
    before calling this function.
    - The function assumes that the 'Proceed to Checkout' button
    is visible and clickable from the cart page.
    - The function navigates through the checkout process by
    clicking 'Next' on the Shipping page and 'Place Order' on the
    Review & Payments page.
    - If the 'Place Order' button is not immediately visible, a
    delay is included to allow dynamic elements to load.
    """
    import asyncio

    await page.goto("/")
    await page.get_by_role("link", name="My Cart").click()
    await asyncio.sleep(5)
    await page.get_by_role("button", name="Proceed to Checkout").
    click()
    await asyncio.sleep(5)
    await page.get_by_role("button", name="Next").click()
    await asyncio.sleep(5)
    await page.get_by_role("button", name="Place Order").click()
    await asyncio.sleep(5)
```

Here is a skill that was learned for the shopping website that represents applying a discount code to the items in the cart.

express_checkout

```
async def apply_discount_code(page, discount_code):
    """
    Apply a discount code during the checkout process.

    Args:
        page: The Playwright page object to perform actions on.
        discount_code (str): The discount code to apply.

    Usage Log:
    - Attempted to apply 'DISCOUNT10', but it was invalid. Ensure valid
    discount codes are used.

    Note:
```

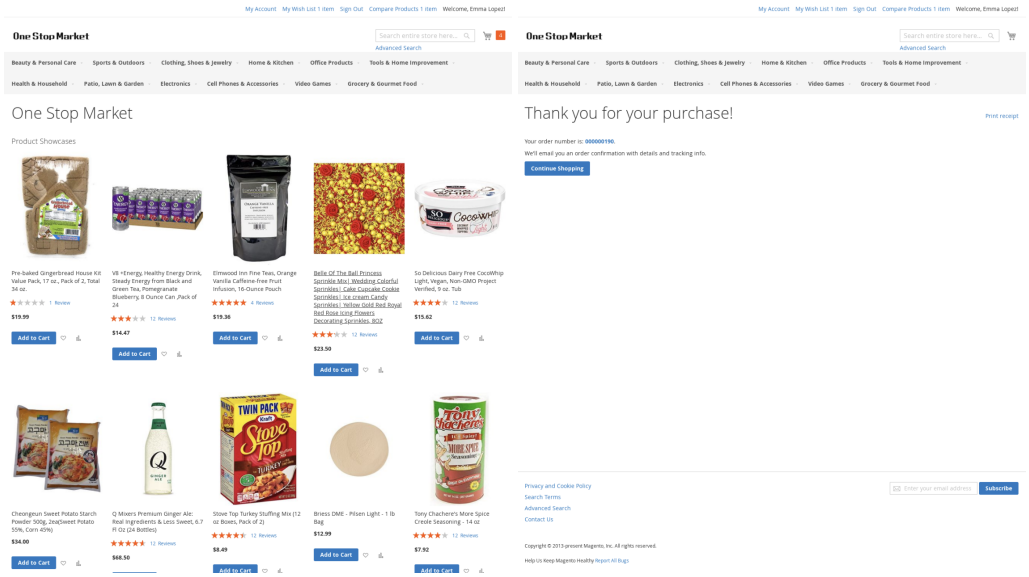

Figure 4: Express Checkout API execution, before (left) and after (right).

```
        - The function navigates to the checkout page to ensure the correct initial
          state.
        - Handles invalid discount codes by checking for error messages or lack of
          discount application.
        """
        await page.goto("/checkout/#payment")
        await page.get_by_role("tabpanel", name="Apply Discount Code").get_by_role(
            "textbox", name="Enter discount code"
        ).fill(discount_code)
        await page.get_by_role("tabpanel", name="Apply Discount Code").get_by_role(
            "button", name="Apply Discount"
        ).click()
```

E.1.2 Debugging Process Fixing a Skill's Implementation

Here, we demonstrate that the agent is capable of fixing DOM resolution errors, by making specific tweaks to problematic sections of code. After the following trajectory, the agent makes a patch to the knowledge base.

Error Encountered for `delete_merged_branches`

<state 0>
URL: /byteblaze/accessible-html-content-patterns/-/merge_requests/14

[Omitted]
</state 0>

<reasoning>
1. I need to delete merged branches in the specified project 'byteblaze/accessible-html-content-patterns'.
2. I can utilize the `delete_merged_branches` function which is predefined to handle this operation effectively in the GitLab UI.

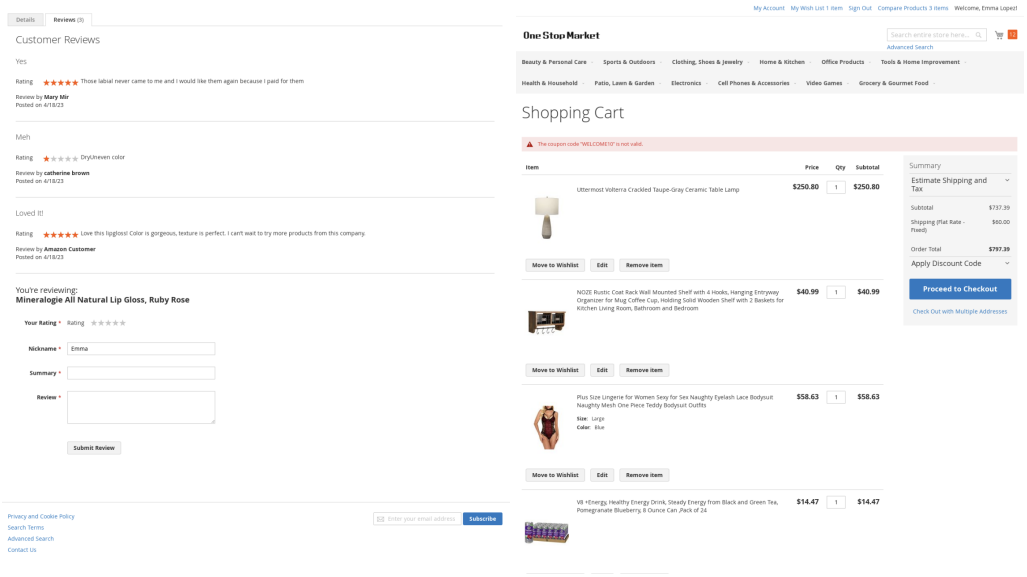

Figure 5: Apply Discount Code API execution. Before, the agent is on a random page (left), and after, the agent successfully navigates to the shopping cart and enters a discount code, `WELCOME10` (right).

3. Since the page indicates that I am already within the correct project context, I can proceed directly with invoking the function using the provided project path.
</reasoning>

<code>

```
async def act(page):
    # Call the function to delete merged branches
    await delete_merged_branches(page, "byteblaze/accessible-html-content-patterns")
```

</code>

We found a bug in one of the APIs, and were able to make the following recoveries. For each RECOVERY comment, follow the instructions.

<exception>
'Error': Locator.fill: Error: strict mode violation: get_by_role("textbox") resolved to 2 elements:
1) <input id="search" type="text" name="search" autocomplete="off" placeholder="Search GitLab" data-qa-selector="search_box" class="form-control gl-form-input gl-search-box-by-type-input" /> aka get_by_placeholder("Search GitLab")
2) <input type="text" id="__BVID__158" autocomplete="off" aria-labelledby="input-label" data-qa-selector="delete_merged_branches_input" class="gl-form-input gl-mt-2 form-control gl-form-input-sm" /> aka locator("[id=\"__BVID__158\"]")

Call log:
</exception>

<state 1>

URL: /byteblaze/accessible-html-content-patterns/-/branches

[Omitted]
</state 1>

<reasoning>
The page currently displays a dialog that asks for confirmation to delete all merged branches. It requires typing 'delete' into an input box. After typing the confirmation text, the "Delete merged branches" button can be clicked to proceed. I will fill the textbox with 'delete' and then click the confirmation button to complete the task.
</reasoning>

<code>

```
async def act(page):
    # Fill the confirmation textbox with 'delete'
    await page.get_by_role("dialog", name="Delete all merged
    branches?").get_by_role("textbox").fill("delete")

    # Click the "Delete merged branches" button to confirm
    deletion
    await page.get_by_role("dialog", name="Delete all merged
    branches?").get_by_role("button", name="Delete merged branches
    ").click()
```

</code>

<state 2>
URL: /byteblaze/accessible-html-content-patterns/-/branches

[Omitted]
</state 2>

<terminate_with_result>
The merged branches have been successfully deleted in the 'byteblaze/accessible-html-content-patterns' project.
</terminate_with_result>

<state 3>
URL: /byteblaze/accessible-html-content-patterns/-/branches

[Omitted]
</state 3>

Here is the fix the agent made: removing the ambiguous DOM selector for "textbox", and replacing it with one that first selects the dialog window that contains the textbox, and then the textbox element within it.

delete_merged_branches

```
async def delete_merged_branches(page, project_path):
    """
    Automates the deletion of merged branches in a specified GitLab project.

    This function navigates to the branches page of the specified project and
     deletes all merged branches
    by interacting with the confirmation dialog that appears.

    Args:
        page: The Playwright page object.
        project_path: The path of the project in GitLab (e.g., 'byteblaze/
    accessible-html-content-patterns').

    Usage log:
    - Successfully used to delete merged branches in the 'a11y-syntax-
    highlighting' project by navigating to the branches page,
      clicking the 'Delete merged branches' button, and confirming the action
    by typing 'delete'.
    - Ensure the project path is correctly specified to match the URL structure
      in GitLab.
    - The function assumes the user is logged in and has access to the
    specified project.
    - Resolved an issue with strict mode violation by specifying the correct
    textbox within the dialog.
    """
    await page.goto(f"/{project_path}/-/branches")
    await page.get_by_role("button", name="Delete merged branches").click()
++   await page.get_by_role("textbox")
--   await page.get_by_role("dialog", name="Delete all merged branches?").
    get_by_role(
--       "textbox"
--   ).fill("delete")
    await page.get_by_role("dialog", name="Delete all merged branches?").
    get_by_role(
        "button", name="Delete merged branches"
    ).click()
    await asyncio.sleep(2)
```

E.2 Failure Cases and Limitations

We observe some failure cases, which seem to be caused by gaps in LLM reasoning capabilities for certain situations.

E.2.1 Criteria for Verifying Success

Because our criteria for a function to be "verified" was to have it be called without producing an exception, we found that occasionally, malfunctioning APIs could be marked as verified simply because they silenced all exceptions that could have occurred. This represents a measure for evaluation having unintended consequences.

In the following example, a function was misused, and it was unable to resolve the necessary DOM elements given the page state. However, instead of improving the function's signature or adding a check to ensure the function was called correctly, the LLM adds "if" statements to simply avoid any of the atomic actions from producing an error. While this does reduce the number of exceptions, it does not improve the robustness of the API.

update_product_quantity

```
async def update_product_quantity(page, product_name, new_quantity):
    """
    Update the quantity of a specified product in the Magento Admin Panel's
     Advanced Inventory settings.

    This function navigates to the product's edit page, opens the Advanced
     Inventory dialog,
    and updates the quantity to the specified value. It includes proactive
     checks for element availability.

    Initial UI State:
    - The user must be logged into the Magento Admin Panel.
    - The function assumes the product edit page is accessible via the '/admin/
    catalog/product/edit/name/{product_name}/' URL.

    Args:
    page (Page): The Playwright page object representing the browser page.
    product_name (str): The name of the product to update.
    new_quantity (int): The new quantity to set for the product.

    Usage Log:
    - Successfully updated the quantity for 'Arcadio Gym Short' to 50.
    - Adjusted the quantity for 'Yoga Mat' to 100 without issues.
    - Encountered a timeout error when trying to update 'Sprite Yoga Mat'.
    Consider checking the availability of the 'Advanced Inventory' button.
    """
    import re
    import asyncio

    await page.goto(f"/admin/catalog/product/edit/name/{product_name}/")
    advanced_inventory_button = page.get_by_role("button", name="Advanced
    Inventory")
    if await advanced_inventory_button.count() > 0:
        await advanced_inventory_button.click()
    else:
        print("Advanced Inventory button not found.")
        return
    qty_input = page.get_by_role("textbox", name="[GLOBAL] Qty")
    if await qty_input.count() > 0:
        await qty_input.fill(str(new_quantity))
    else:
        print("Quantity input not found.")
        return
    done_button = page.get_by_role("button", name="Done")
    if await done_button.count() > 0:
        await done_button.click()
    else:
        print("Done button not found.")
        return
    save_button = page.get_by_role("button", name="Save")
    if await save_button.count() > 0:
        await save_button.click()
    else:
        print("Save button not found.")
        return
```

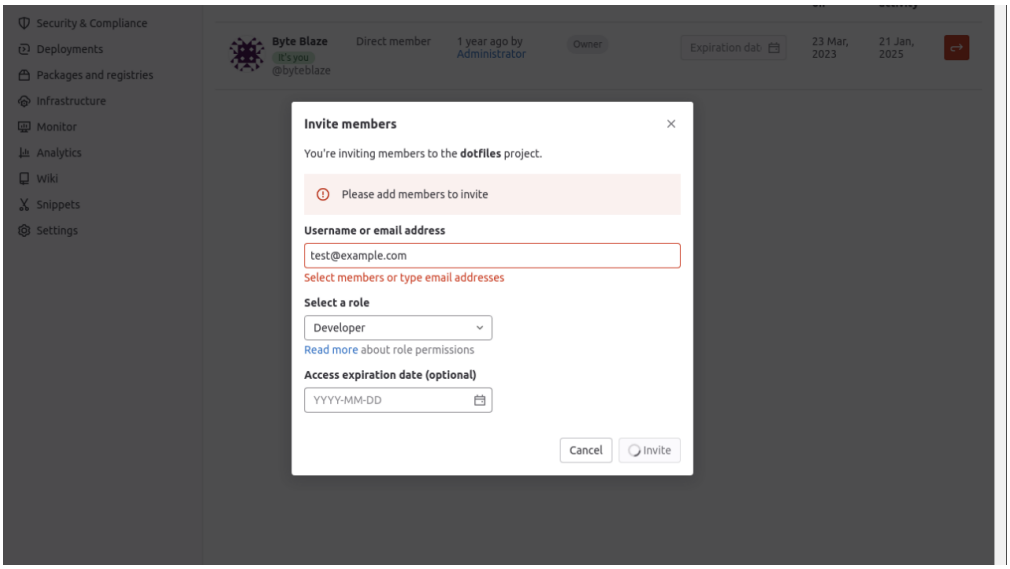

Figure 6: The agent is unable to successfully verify that the "add collaborator to project" works because it does not have access to a list of emails that can appropriately be used as test cases.

E.2.2 Creating Functional Test Cases

Some skills require interacting with data stored on the website, and their success is conditioned not on whether the UI is accurately interacted with but rather whether the function inputs are realistic. In the following example, the agent attempts to learn how to add collaborators to a Gitlab repository, but is unable to because it does not have a source of real test input data. Future work could explore automatically generating test cases for agents for successful evaluation in dynamic websites.

E.2.3 Ceiling of Agent Capabilities

request_quote_for_multiple_listings

```
async def request_quote_for_multiple_listings(page, make, model, first_name,
    last_name, email):
    """
    Request a quote for multiple car listings on Cars.com by navigating through
     listings and submitting the quote request form.

    This function navigates to the 'Cars for Sale' section, applies filters for
     the specified make and model,
    and iterates through the listings to request quotes. It manually navigates
    between listings using their URLs.

    Args:
        page (Page): The Playwright page object.
        make (str): The car make to filter by.
        model (str): The car model to filter by.
        first_name (str): The first name to fill in the contact form.
        last_name (str): The last name to fill in the contact form.
        email (str): The email to fill in the contact form.

    Usage log:
```

```
    - Successfully navigated to the 'Cars for Sale' section, applied filters,
     and requested quotes for multiple listings.
    - Encountered issues with automatic navigation using 'Next car' link,
     resolved by manual URL navigation.
    """
    import asyncio

    await page.goto("https://www.cars.com/shopping/")
    await page.wait_for_load_state("networkidle")
    await close_overlays(page)
    await page.get_by_role("combobox", name="Make").select_option(make)
    await page.get_by_role("combobox", name="Model").select_option(model)
    show_matches_button = page.get_by_role("button", name="Show matches")
    if await show_matches_button.count() > 0:
        await show_matches_button.click()
    else:
        print("Show matches button not found.")
        return
    await page.wait_for_load_state("networkidle")
    for _ in range(5):
        await page.get_by_role("textbox", name="First name").fill(first_name)
        await page.get_by_role("textbox", name="Last name").fill(last_name)
        await page.get_by_role("textbox", name="Email").fill(email)
        await page.get_by_role("combobox", name="Subject").select_option(
            "Get a price quote"
        )
        submit_button = page.get_by_role("button", name="Email")
        if await submit_button.count() > 0:
            await submit_button.click()
        else:
            print("Submit button not found.")
            return
        await asyncio.sleep(2)
        next_car_link = page.get_by_role("link", name="Next car")
        if await next_car_link.count() > 0:
            await next_car_link.click()
            await page.wait_for_load_state("networkidle")
        else:
            print("Next car link not found. Navigation stopped.")
            break
```

E.2.4 Fail to call API

The agent does not call available APIs even when they are generated during exploration. As shown in Figure 7, for the task "Save a hamburger recipe," the agent should first call `search_recipes_by_cuisine_type('hamburger')` to obtain recipe details and then select the print option to save the recipes. However, the agent fails to call this API during the task execution, indicating a gap between exploration and execution phases, suggesting the need for improving the agent's policy to better utilize generated APIs.

`search_recipes_by_cuisine_type`

```
async def search_recipes_by_cuisine_type(page, cuisine_type):
    """
    Searches for recipes based on a specified cuisine type on Cookpad.

    This function automates the process of searching for recipes by entering a
     cuisine type
    into the search functionality on the Cookpad homepage.
```

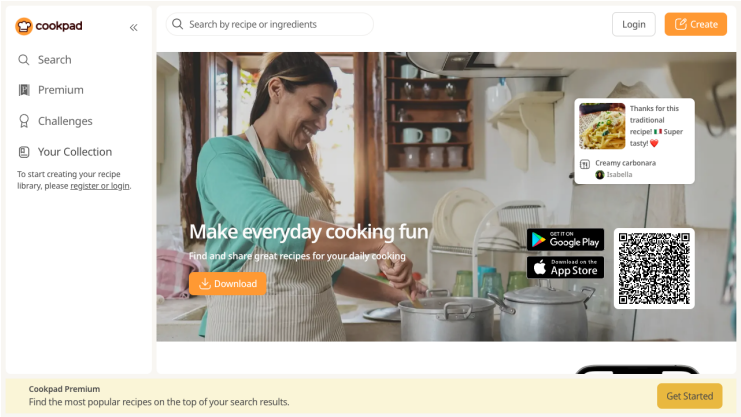

Figure 7: Screenshot of cookpad.com. The agent fails to call `search_recipes_by_cuisine_type('hamburger')` to obtain recipe details.

```
    Args:
        page: The Playwright page object to interact with.
        cuisine_type (str): The cuisine type to search for, e.g., 'Italian', '
    Mexican', 'Korean', 'Chinese', 'American'.

    Usage log:
    - Successfully used to search for 'Italian' cuisine recipes, displaying the
      correct results.

    """
    import re

    await page.get_by_role("textbox", name="Search by recipe or ingredients").
    fill(
        cuisine_type
    )
    search_button = page.get_by_role("button", name=re.compile("Search", re.
    IGNORECASE))
    if await search_button.is_visible():
        await search_button.click()
    else:
        await page.keyboard.press("Enter")
```

E.2.5 Wrong Parameter

In some cases, the agent correctly identifies the appropriate API but selects incorrect parameters. For example in Figure 8, in the task "Browse recipes for gluten-free chocolate chip cookies that can be made without nuts," the agent incorrectly uses the parameter `search_recipes_by_ingredients(page, 'chocolate chip, -nuts')`, whereas the correct parameter should be `search_recipes_by_ingredients(page, 'chocolate chip without nuts')`. This indicates that the agent needs better parameter selection logic to enhance performance.

`search_recipes_by_ingredients`

```
async def search_recipes_by_ingredients(page, ingredients):
    """
```

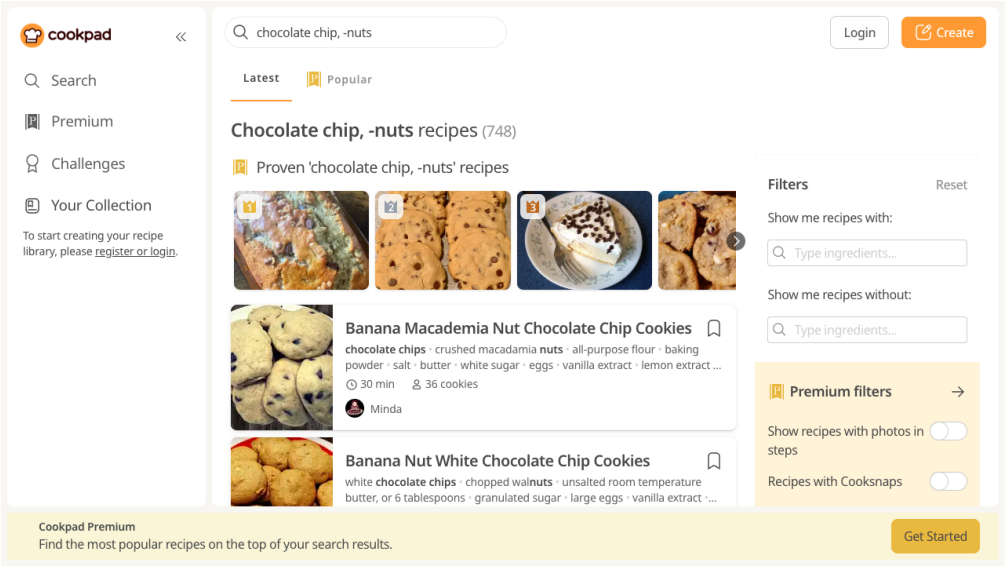

Figure 8: Screenshot of cookpad.com. The agent incorrectly uses the parameter `search_recipes_by_ingredients(page, 'chocolate chip, -nuts')` instead of the correct parameter `search_recipes_by_ingredients(page, 'chocolate chip without nuts')`

```
    Searches for recipes using specified ingredients on Cookpad.

    This function automates the process of searching for recipes based on a
     list of ingredients
    by interacting with the search functionality on the Cookpad homepage.

    Args:
        page: The Playwright page object to interact with.
        ingredients (str): A comma-separated string of ingredients to search
     for, e.g., 'chicken, rice, broccoli'.

    Usage log:
    - Successfully used to search for recipes with 'chicken, rice, broccoli',
     displaying the correct results.
    - Successfully used to filter recipes by seasonal ingredients 'pumpkin,
     apple, squash', displaying the correct results.

    """
    import re

    await page.get_by_role("textbox", name="Search by recipe or ingredients").
    fill(
        ingredients
    )
    search_button = page.get_by_role("button", name=re.compile("Search", re.
    IGNORECASE))
    if await search_button.is_visible():
        await search_button.click()
    else:
        await page.keyboard.press("Enter")
```

```python
async def apply_multiple_filters(page, make, model, distance, zip_code, max_price):
  # Apply multiple filters on the Cars.com website including make, model, distance, ZIP code,
and price range.
  await page.goto("https://www.cars.com/shopping/")
  await close_overlays(page)
  await filter_cars_by_make_and_model(page, make, model)
  await refine_search_by_distance_and_zip(page, distance=distance, zip_code=zip_code)
  if max_price and max_price.strip():  price_select = page.locator('#make-model-max-price')
  if await price_select.count() > 0:  await price_select.select_option(value=str(max_price))
  else: print("Price selector not found")
  search_button = page.get_by_role("tabpanel", name="Make").locator("spark-button[data-
searchtype='make']")
  if await search_button.count() > 0:  await search_button.click()
  else:  print("Search button not found in make panel")
```

Figure 9: An example of compositional API that calls multiple other APIs.

E.3 Compositional API

E.4 Website Roadmap API

```python
async def navigate_to_cars_for_sale(page):
  # Navigate to the 'Cars for Sale' section on the Cars.com website, which contains car
sales information.
  import asyncio
  await page.goto("https://www.cars.com/")
  cars_for_sale_link = page.get_by_role("navigation", name="main menu").get_by_role(
                "link", name="Cars for Sale")
  if await cars_for_sale_link.count() > 0:
    await cars_for_sale_link.click()
    await asyncio.sleep(5)
  else: print("Cars for Sale link is not visible.")
```

Figure 10: An example of roadmap information contained in the synthesized APIs. The docstring of this API contains the corresponding functionality of this API. If the agent is new to this website, this function might requires some exploration to get.