WISE Pendidikan Indonesia

BLOCKCHAIN, ARTIFICIAL INTELLIGENCE, AND FUTURE RESEARCH

ISSN: 3089-6975

LLM-Agent-Style Automated Usability Testing on MiniWoB ++: A Reproducible Chunked Full-Run with ReAct, Plan-Execute, and Self-Reflect Policies

Qi Xin*

To cite this article: Q. Xin, "LLM-Agent-Style Automated Usability Testing on MiniWoB ++: A Reproducible Chunked Full-Run with ReAct, Plan-Execute, and Self-Reflect Policies," *Blockchain, Artif. Intell. Futur. Res.*, vol. 2, no. 1, pp. 56–82, 2026.
DOI: https://doi.org/10.70211/bafr.v2i1.384

To link to this article:

Published online: 30 May 2026

Submit your article to this journal

View crossmark data

Full Terms & Conditions of access and use can be found at
https://journal.wiseedu.co.id/index.php/bafrjournal/about

OPEN ACCESS

https://doi.org/10.70211/bafr.v2i1.384

Blockchain, Artificial Intelligence, and Future Research Vol 2 No 1 (2026)

Research Article

LLM-Agent-Style Automated Usability Testing on MiniWoB++: A Reproducible Chunked Full-Run with ReAct, Plan-Execute, and Self-Reflect Policies

Qi Xin*

Received: 5 March 2026	Revised: 18 March 2026	Accepted: 23 May 2026	Online: 30 May 2026

Abstract

Automated usability testing can reduce the cost of repeatedly checking whether a web interface supports reliable and efficient task completion, but existing scripted tests are brittle and many agent evaluations report benchmark scores without translating failures into usability diagnostics. This study asks how three LLM-agent-style strategies—ReAct, Plan-Execute, and Self-Reflect—differ in effectiveness, efficiency, and failure modes when applied to MiniWoB++ tasks, and whether their logged traces can support actionable UI analysis. We conducted a controlled experimental benchmark on 130 MiniWoB++ web tasks, running each strategy once under a fixed seed with the same deterministic DOM-grounded controller, headless Chromium harness, 10-step limit, and 2.0 s episode budget, producing 390 episodes. We analyzed task success, steps, wall-clock time, interaction category, difficulty bins, and failure categories using paired per-task comparisons and descriptive aggregation. Plan-Execute achieved the highest success rate (14.6%, 19/130), compared with 10.8% (14/130) for both ReAct and Self-Reflect; its advantage was most evident in form/transaction and selection tasks, while all strategies performed similarly on simple click/button tasks and failed on drag/scroll tasks. Failure analysis showed that wrong outcomes and element-grounding errors were the dominant bottlenecks, indicating that explicit planning improves coverage only when target elements can be reliably grounded. The findings contribute a reproducible baseline and a usability-oriented failure taxonomy for automated web-agent testing, suggesting that future frameworks should prioritize semantic grounding, plan validation, richer action primitives, and designer-facing diagnostics.

Keywords: Automated Usability Testing; Web Agents; MiniWoB++; ReAct; Planning

Publisher's Note:
WISE Pendidikan Indonesia stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright:
©
2026 by the author(s).
License WISE Pendidikan Indonesia, Bandar Lampung, Indonesia.
This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY 4.0) license (https://creativecommons.org/licenses/by/4.0/).

INTRODUCTION

Automated usability testing is needed because web interfaces change frequently, but teams cannot repeat full human task studies after every UI revision. The problem addressed in this study is not simply whether an automated agent can complete MiniWoB++ tasks, but whether its successes, efficiency, and failures can be interpreted as reproducible usability evidence. Existing scripted tests verify fixed paths, while many web-agent benchmarks report success scores without showing how failures translate into design actions. This gap motivates a controlled comparison of LLM-agent-style policies as synthetic usability probes.

This study has three objectives. First, it compares ReAct, Plan-Execute, and Self-Reflect strategies in terms of effectiveness and efficiency on MiniWoB++. Second, it examines whether performance changes across interaction types and DOM-derived difficulty bins. Third, it analyzes failure categories to determine which errors can guide UI designers toward more discoverable labels, clearer flows, and better support for non-standard widgets.

Usability is commonly defined through effectiveness, efficiency, and satisfaction in a specified context of use [1]. Human task-based testing remains the most direct way to measure these dimensions because evaluators can observe task completion, time-on-task, errors, and user perceptions [2]. However, human studies are expensive to repeat at the frequency of modern UI regression cycles. Automated agents cannot replace subjective satisfaction measures such as SUS [3], but they can repeatedly estimate effectiveness and efficiency and can provide structured traces that indicate likely sources of friction.

Traditional browser automation partially addresses regression testing through tools that replay scripted actions. These tests are useful for verifying critical workflows, but they assume that the intended path is already known and stable. Usability problems often arise when labels are ambiguous, affordances are hidden, or a workflow branches unexpectedly. A language-conditioned agent offers a different form of automation: it receives a goal statement, observes the interface, selects actions, and logs where the attempt succeeds or fails. This makes agent-based testing closer to the question asked in usability work: can the goal be completed with reasonable effort?

MiniWoB and MiniWoB++ provide a suitable controlled setting for this question. World of Bits and MiniWoB introduced browser-based tasks with rewards tied to task completion [5], and MiniWoB++ expanded the suite into diverse web tasks involving clicking, typing, selection, and short multi-step workflows [4]. These environments are deliberately simpler than open-world websites, but that simplicity is useful for isolating interaction failures: a failed episode can often be traced to a specific instruction, element, or action primitive rather than to external content variability.

Recent LLM-agent research adds reasoning and control scaffolds that could improve such interaction. ReAct interleaves reasoning and acting [7], [8]; planning-oriented methods separate goal decomposition from execution [9], [10], [11]; and reflection methods attempt to revise behavior after failure [8]. Prior studies show the promise of these scaffolds, but their results are often influenced by model choice, prompt wording, tool configuration, and stochastic sampling. For usability testing, this is a limitation because teams need repeatable signals and interpretable failure categories, not only a best-case success rate.

This paper therefore evaluates the strategy scaffolds under a deterministic, DOM-grounded controller. This design intentionally limits absolute performance, but it isolates the structure of the policy—reactive, planned, or reflective—from variability in model sampling. The resulting baseline

is useful in two ways: it provides a lower bound for future LLM-driven agents, and it makes failure analysis reproducible enough to support UI-design recommendations.

The study addresses the following research questions: (RQ1) How do ReAct, Plan-Execute, and Self-Reflect differ in task success, steps, and wall-clock time on MiniWoB++? (RQ2) How does performance vary by interaction category and by a reproducible DOM-derived difficulty proxy? (RQ3) Which failure modes dominate, and how can they be mapped to actionable recommendations for automated usability-testing frameworks and UI designers?

The results show that explicit planning improves coverage on structured form and selection tasks, while the implemented reflection fallback does not improve over ReAct. Plan-Execute solves 19 tasks (14.6%) compared with 14 tasks (10.8%) for both ReAct and Self-Reflect. Most remaining failures are wrong-outcome terminations or element-grounding failures, indicating that semantic grounding, plan validation, and richer action primitives are the main bottlenecks. The remainder of the paper reviews the relevant literature, describes the deterministic evaluation harness, presents quantitative and diagnostic results, and discusses implications for agent-based usability testing.

LITERATURE REVIEW

Usability Foundations and Task-Based Measurement

Formal usability definitions emphasize effectiveness, efficiency, and satisfaction within a specified context of use [1]. In practice, usability engineering operationalizes these dimensions through task-based evaluations that quantify task completion and time-on-task, and complements them with heuristic analyses that explain failures in terms of mismatched mental models, weak affordances, and error-prone workflows [2]. Standardized questionnaires such as the System Usability Scale (SUS) provide a validated subjective measure of perceived usability, enabling comparisons across systems and versions, but require human raters and therefore do not transfer directly to fully automated evaluation pipelines [3]. These foundations motivate proxy metrics for automation: an automated "synthetic user" directly estimates effectiveness and efficiency, while diagnostic traces and failure categories approximate the qualitative evidence that human evaluators typically extract.

From Scripted UI Tests to Agentic Interaction

Traditional end-to-end UI regression frameworks verify that a specific interaction script still executes, which catches hard breakages but remains brittle under changes in label text, layout, and alternate workflow branches. Recent work on language-conditioned agents reframes interaction as goal achievement: instead of replaying a fixed script, an agent interprets a natural-language instruction and selects actions conditioned on observations. This framing aligns with usability evaluation, where the objective is not merely that a "happy path" exists, but that the goal is discoverable and achievable with reasonable effort.

Web-Interaction Benchmarks for Learning and Evaluation

World of Bits introduced an open-domain platform for web-based agents, exposing both visual observations and structured information such as the DOM, and provided a benchmark suite that made web interaction amenable to reinforcement learning and imitation learning methods [5]. Building on this direction, MiniWoB-style tasks offer a controlled collection of micro-websites designed to test core interaction primitives (clicking, typing, selecting) and their composition into short workflows; workflow-guided exploration demonstrated how task structure and

demonstrations improve sample efficiency under sparse rewards in these environments [4]. MiniWoB++ further standardized and expanded the environment set and, through community maintenance, has been packaged as a Gymnasium-compatible library to support reproducible evaluation and toolchain integration [6]. Because these tasks are deterministic under a fixed seed and include explicit success signals, they support large-scale measurement of effectiveness and efficiency and enable failure tracing to concrete UI elements.

Reasoning-Acting Scaffolds for Language-Model Agents

Prompting methods such as chain-of-thought elicitation showed that generating intermediate reasoning steps improves performance on multi-step problems, motivating agent designs that externalize reasoning as part of the control loop [12]. ReAct operationalized this idea by interleaving reasoning traces with environment actions, using observations to ground subsequent decisions and to reduce hallucinated plans in interactive settings [7]. Subsequent work explored planning and search-like inference at the prompting level: Plan-and-Solve separates explicit planning from execution to reduce missing-step errors in multi-stage reasoning [9], Tree of Thoughts generalizes chain-of-thought to explore multiple candidate "thought" branches and select promising trajectories [11], and ReWOO decouples reasoning from tool observations to improve efficiency in augmented language-model agents [10]. Tool use is another recurring theme: Toolformer introduced a training approach that teaches language models to decide when to invoke external tools and how to incorporate tool outputs, providing a mechanism for scalable tool-augmented behavior beyond hand-written prompting templates [13]. Together, these works motivate the three strategy families evaluated in this paper (reactive, planning-based, and reflection-based) as reusable scaffolds for UI interaction.

Broader Web-Agent Environments and Multimodal Interaction

While MiniWoB++ emphasizes isolated UI primitives in a controlled micro-internet, newer benchmarks evaluate agents on longer-horizon, multi-page workflows and realistic website structures. WebShop provides a simulated e-commerce site with natural-language shopping instructions and large-scale product data, requiring agents to combine language understanding with strategic navigation [14]. WebArena constructs a reproducible environment around multiple websites and tasks that resemble real user journeys and reports baseline evaluations of autonomous agents in this setting [15]. BrowserGym provides an ecosystem that unifies web-agent benchmarks under a standardized interface and experiment-management stack intended to reduce fragmentation and support reproducible comparisons across environments [16], [17], [18], [19], [20], [21]. VisualWebArena extends these benchmarks with visually grounded web tasks that require multimodal perception and reports quantitative evaluations that contrast text-only and multimodal agent behaviors [22].

Learning From Feedback and The Role of Interfaces

Agent performance in interactive environments depends not only on the core model but also on the feedback and interface through which the agent interacts. Reinforcement learning from human preferences introduced a general framework for learning reward models from comparative feedback, enabling optimization when explicit reward functions are difficult to specify [20]. Instruction-following language models trained with human feedback demonstrate that alignment

and helpfulness improve by combining supervised demonstrations with preference-based optimization, providing a route to more reliable behaviors under natural-language goals [19]. WebGPT studied browser-assisted question answering in a text-based web-browsing environment optimized with human feedback and explicit citation collection, illustrating how browsing tools and feedback loops can be combined for grounded interaction [21]. Complementing these learning-centric approaches, SWE-agent emphasized that agents benefit from purpose-built agent-computer interfaces (ACIs) and empirically measured how interface design affects agent task success in software-engineering workflows [18]. This result directly connects to usability: if interface design affects agent performance, then agent traces serve as a scalable probe of interaction difficulty, and agent-facing observability improvements (e.g., accessible labels and consistent action affordances) support both agent and human usability.

Synthesis and Gap

Existing benchmarks and agent frameworks provide strong components controlled web interaction tasks [4], [5], [6], goal-conditioned web environments [14], [15], [16], multimodal extensions [22], and prompting-based scaffolds for reasoning, planning, and reflection [7], [8], [9], [10], [11], [12], [13] but they typically report interaction success primarily as a benchmark score rather than as a usability-oriented measurement aligned with effectiveness and efficiency [1], [2], [3]. In addition, many agent evaluations depend on proprietary models and non-deterministic sampling, which complicates exact reproducibility and fine-grained failure attribution. The present study builds on these foundations by (i) treating agent task success, steps, and time as direct proxies for usability effectiveness and efficiency, and (ii) emphasizing reproducible, strategy-level comparisons with structured failure taxonomies that translate interaction traces into actionable UI diagnostics for design and engineering teams.

METHODS

Figure 1 illustrates the benchmark pipeline and logging interface used in this study.

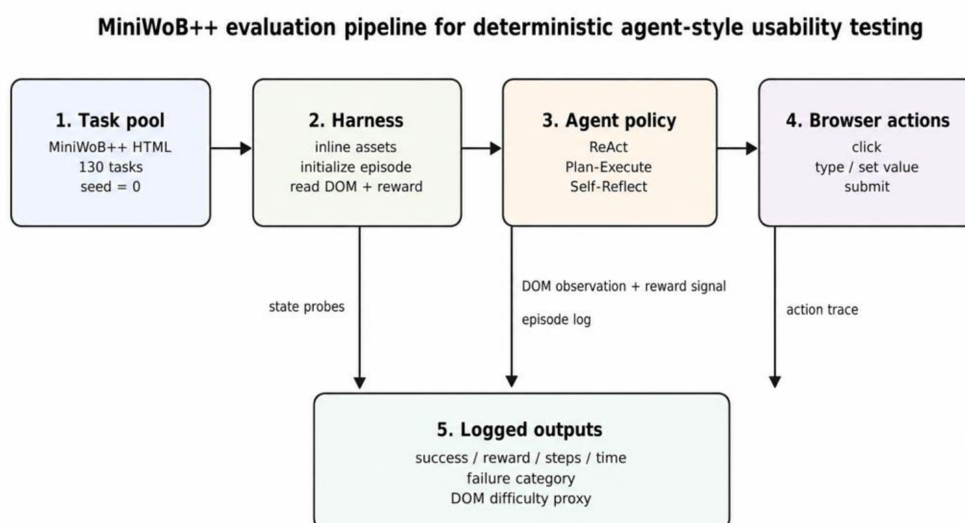


Figure 1. Overview of the MiniWoB++ evaluation pipeline (task loading, harness, agent policy, actions, and logged metrics).

This section details the experimental design, environment setup, agent initialization, action interface, logging procedure, and usability-oriented metrics used in the study. The evaluation is a controlled experimental benchmark: each of the 130 MiniWoB++ tasks is run once under seed=0 for each of three agent strategies, yielding 390 total episodes. All strategies use the same browser harness, DOM-grounding primitives, action budget, and failure taxonomy, so differences are attributable to policy structure rather than environment configuration.

MiniWoB++ tasks and episode structure. MiniWoB++ provides web pages that emulate common user-interface patterns, including buttons, lists, date pickers, text fields, and short multi-step forms. Each task contains a natural-language instruction in the #query element and a JavaScript controller that updates global completion and reward variables. At the beginning of each episode, the harness sets the deterministic task seed, starts the task controller, and then allows the agent to interact until success, failure, a step limit, or a time budget is reached.

Browser and environment setup. Each task is executed in a headless Chromium browser controlled through Playwright. To avoid external network variation, the harness builds a self-contained HTML page by inlining local JavaScript, CSS, and image assets before loading the task. The browser viewport is fixed at 1024×768, and each episode uses a maximum of 10 agent actions and a 2.0-second wall-clock budget. After every action, the harness waits briefly for DOM updates and reward computation before reading the next state. The seed is defined by the task name and seed value (task_name|0), making the episode initialization reproducible on the same codebase and browser configuration.

Agent initialization and shared grounding. For every task-agent pair, the agent is initialized with no memory from previous tasks and receives only the current task instruction and DOM state. The three policies share the same text normalization, target extraction, element search, and action execution functions. This shared controller is important because it isolates the effect of the strategy scaffold: ReAct, Plan-Execute, and Self-Reflect differ in how they order decisions, not in what elements they are allowed to see or what browser actions they can perform.

Observation interface. At each step, the agent observes the instruction text extracted from #query and a simplified DOM view. The harness also records initial DOM features for analysis: the number of visible clickable elements, visible input/select elements, and total visible elements. These features are not used to change the agent policy; they are logged to construct a post-hoc difficulty proxy and to identify whether failures are associated with denser interfaces or non-standard widgets.

Action interface. Each policy emits one of three action types. A click action selects a visible clickable element by matching the instruction target to normalized inner text, value, aria-label, id, or related label text. A type/set-value action selects a visible input, textarea, or select element and fills a value extracted from the instruction; select controls are handled by matching option text or value. A submit action clicks a visible control whose text suggests completion, such as submit, ok, search, send, or confirm. The restricted action interface supports common MiniWoB++ interactions while making failures on unsupported gestures, such as dragging and scrolling, easy to interpret.

ReAct policy. The ReAct policy implements a reactive interpret-then-act loop. At each step, it parses the instruction for typing, clicking, or submission cues and immediately chooses the first grounded action that matches those cues. It attempts value entry when the instruction contains type/enter language or simple key-value patterns, attempts target clicks when click-like verbs are present, and otherwise searches for a submit-like control. This deterministic version follows the

ReAct idea of interleaving reasoning and action, but the reasoning is implemented through rule-based parsing rather than generated natural-language chains of thought.

Plan-Execute policy. The Plan-Execute policy first converts the instruction into a short ordered plan. It extracts key-value pairs, quoted strings, and likely targets; transforms them into fill, click, and submit subgoals; and then executes those subgoals sequentially. Grounding is performed at execution time rather than only at plan creation, so a later step can use the updated DOM after a field is filled or a widget changes. The expected benefit is better handling of tasks that require coordinated sequences, such as entering a value and then confirming it.

Self-Reflect policy. The Self-Reflect policy uses the same grounding functions but adds a lightweight recovery mechanism. It applies Plan-Execute when the instruction appears structured and ReAct otherwise. After each step, it computes a state hash from DOM counts and instruction text; if the hash does not change across steps, the policy marks stagnation and triggers a fallback sequence that first tries submit and then tries the first visible clickable element. This mechanism tests whether simple reflection-style recovery can improve robustness without introducing stochastic sampling.

Usability metrics. Four metrics are logged for every episode. Task success records whether the environment terminates with positive reward and represents effectiveness. Steps records the number of browser actions before termination or budget exhaustion and represents interaction effort. Wall-clock time records elapsed execution time and represents an efficiency proxy under a fixed harness. Failure reason records why an unsuccessful episode ended and provides diagnostic evidence for interpreting friction. These metrics deliberately focus on effectiveness and efficiency because a synthetic agent cannot directly report subjective satisfaction.

Failure categorization. The raw failure_reason string is recorded at the point of failure, for example negative_reward, element_not_found:no_match, element_not_found:no_targets, submit_not_found:no_submit, step_limit, no_action, time_budget, or system_error. These raw labels are then mapped deterministically into broader categories: Success, WrongOutcome, ElementNotFound, InputNotFound, SubmitNotFound, StepLimit, NoAction, TimeBudget, SystemError, and Other. The raw labels support debugging, while the broader categories support aggregation across tasks and comparison across agents.

Use of failure categories for UI design insights. The taxonomy is designed to be actionable rather than merely descriptive. ElementNotFound and InputNotFound suggest that labels, aria attributes, or field associations are not discoverable by the grounding layer and therefore should be audited for clarity and accessibility. SubmitNotFound suggests that completion affordances may be unclear or inconsistently named. WrongOutcome suggests that the agent found a plausible but incorrect path, which points to ambiguous controls, duplicate labels, weak feedback, or missing validation. StepLimit and NoAction indicate flows where the next action is not readily discoverable under the available observation and action model. In this way, the same logs used for benchmarking can be converted into design triage reports.

Data processing and analysis. All episodes are appended to a CSV file with task name, seed, agent identifier, instruction, success flag, reward, steps, wall-clock time, DOM features, raw failure reason, and failure category. The analysis aggregates results overall, by coarse interaction category, and by five equal-frequency difficulty bins computed from a weighted combination of visible controls and instruction length. Paired per-task success vectors are compared using exact McNemar

tests, which directly evaluate whether one strategy solves tasks that another fails under the same deterministic task instance.

Table 1. Benchmark and evaluation configuration

0	1
Dataset	MiniWoB++ (miniwob package tasks)
# tasks	130
# agents	3
Agents	ReAct, PlanExecute, SelfReflect
# seeds per task	1
Seeds	0
Total episodes	390
Browser	Headless Chromium via Playwright
Max steps/episode	10
Wall-clock budget/episode	2.0 s
Observation signals	Instruction (#query), DOM counts, WOB_DONE_GLOBAL, WOB_REWARD_GLOBAL
Action space	Click (visible clickable elements), Type/Set value (inputs/selects), Submit heuristics

Table 2. Agent strategy scaffolds and deterministic instantiations used in this paper

Agent strategy	Decision procedure (this work)	Expected strength
ReAct	Iterative interpret→act loop driven by instruction keywords; greedy click/submit; limited form parsing.	Low compute; strong on single-step click tasks.
PlanExecute	Constructs an explicit plan: (fill key–value pairs / quoted strings)→(click targets)→(submit); executes sequentially.	Best on multi-field forms and selection tasks; higher wrong-outcome risk.
SelfReflect	Runs single-step PlanExecute/ReAct, detects stagnation via state hash, triggers fallback actions (submit/first clickable).	More exploration under uncertainty; slower, did not improve success in this run.

RESULTS AND DISCUSSION

This section reports quantitative performance across the full MiniWoB++ suite and interprets the findings as signals for automated usability testing. Unless stated otherwise, results are computed over 130 tasks evaluated once each per agent under seed=0 (Table 1).

Overall effectiveness and efficiency. Table 3 summarizes the primary outcomes. Plan-Execute attains the highest effectiveness, solving 19 of 130 tasks (14.6%). ReAct and Self-Reflect

each solve 14 tasks (10.8%). The mean number of executed actions per task is 1.32 for Plan-Execute, 2.22 for ReAct, and 2.21 for Self-Reflect (Figure 4). Mean wall-clock times are 0.202s, 0.270s, and 0.347s respectively (Figure 5). These efficiency values are dominated by failures that reach the step or time budget; when conditioning on success, all strategies complete successful tasks in approximately one to two actions and well under one second (Table 3).

Table 3. Overall effectiveness and efficiency metrics by agent (seed=0)

Agent	Tasks solved	Success rate (%)	Mean steps (all)	Mean time (s, all)	Mean steps (success)	Mean time (s, success)
ReAct	14	10.7692	2.2154	0.2704	1.0	0.2058
PlanExecute	19	14.6154	1.3231	0.2023	1.2632	0.2002
SelfReflect	14	10.7692	2.2077	0.347	1.0	0.2352

Figure 2 and Figure 3 visualize step and time efficiency for all episodes and for successful episodes only.

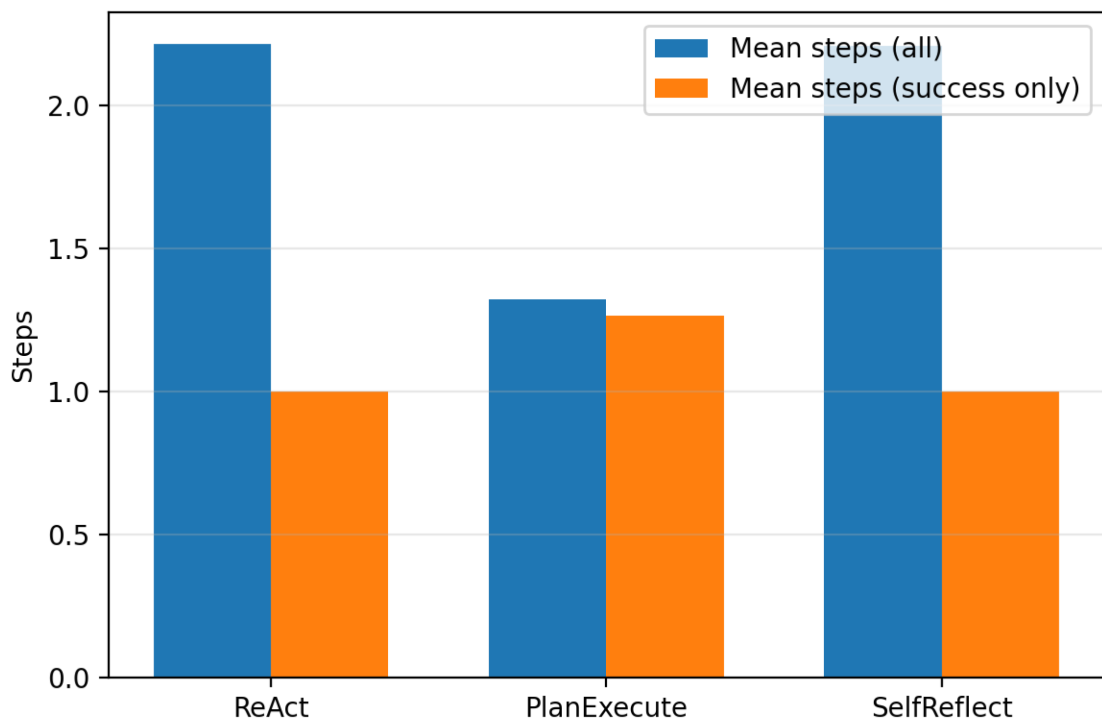


Figure 2. Mean steps per episode (all episodes) and mean steps conditional on success

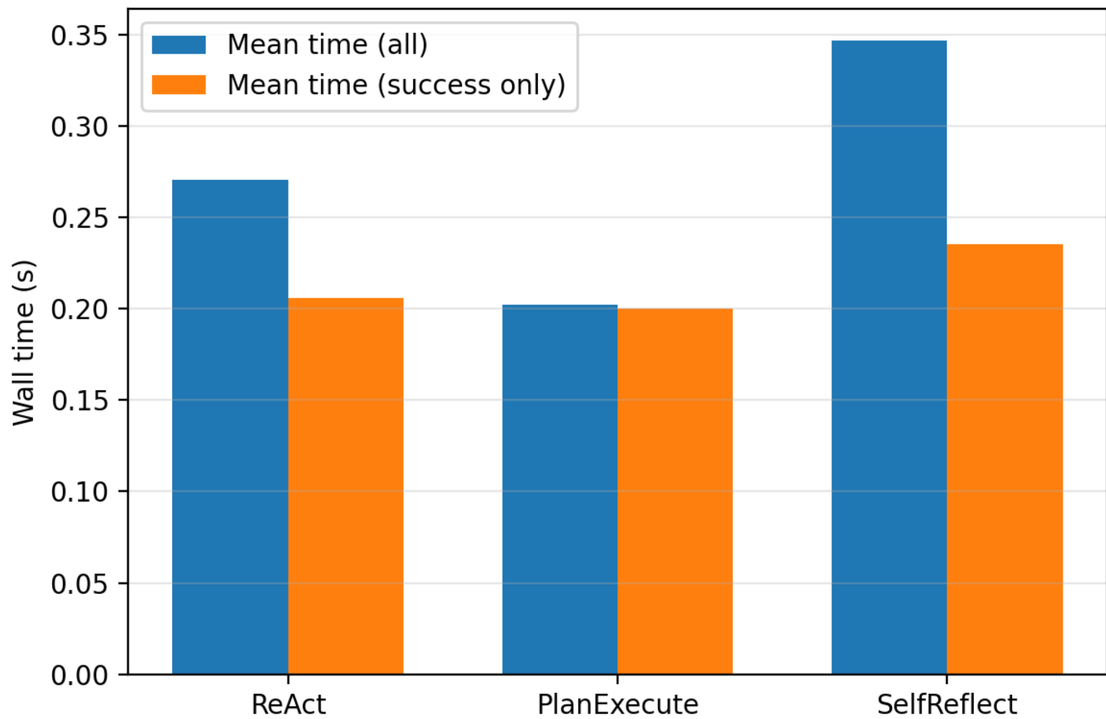


Figure 3. Mean wall-clock time per episode (all episodes) and mean wall-clock time conditional on success

Task overlap analysis clarifies the source of Plan-Execute's advantage. The set of tasks solved by ReAct is identical to the set solved by Self-Reflect in this run. Plan-Execute solves all 14 tasks solved by the other strategies and additionally solves 5 tasks that neither ReAct nor Self-Reflect solves: choose-date-easy, choose-date-medium, enter-text, enter-text-dynamic, and use-autocomplete (Table 8). Each of these additional tasks requires value selection, value entry, or a value-setting step followed by confirmation, aligning with the plan-first control structure.

Interpretation for usability testing. From a usability-evaluation perspective, a higher success rate indicates that the interface supports goal completion under the agent's interpretation model; a lower step/time footprint indicates efficient interaction given the agent's control capabilities. In this benchmark, the absolute success rates remain modest (<15%), which means the agent strategies are primarily useful as stress tests: they identify a narrow set of flows that are robust to simple automation and a broad set that break under even minimal uncertainty. The diagnostic value therefore lies in the failure traces and in differential success across interaction types rather than in aggregate success alone.

Performance by interaction category. To connect outcomes to UI patterns, we categorize tasks using name-based interaction cues (click/button, selection, form/transaction, drag/scroll, other). The category breakdown (Table 4) shows a consistent hierarchy. Click/button tasks are the easiest for all strategies: each agent succeeds on 32.3% of these tasks (Figure 3). Form/transaction tasks—tasks that require filling one or more fields and potentially submitting—are substantially harder. Plan-Execute reaches 8.8% success on this category, while ReAct and Self-Reflect reach 2.9%. Selection tasks (lists, dropdowns, date pickers, and similar controls) are solved only by Plan-Execute in this run (25.0% success vs. 0.0%). Drag/scroll tasks are unsolved by all agents (0.0%),

which matches the fact that the implemented action space does not include drag gestures or scroll-specific strategies.

Table 4. Success rate by interaction category (seed=0)

category	tasks	ReAct_success_rate	PlanExecute_success_rate	SelfReflect_success_rate
Click/Buttons	31	0.3226	0.3226	0.3226
Selection	8	0.0	0.25	0.0
Form/Transaction	34	0.0294	0.0882	0.0294
Other	39	0.0769	0.1026	0.0769
Drag/Scroll	18	0.0	0.0	0.0

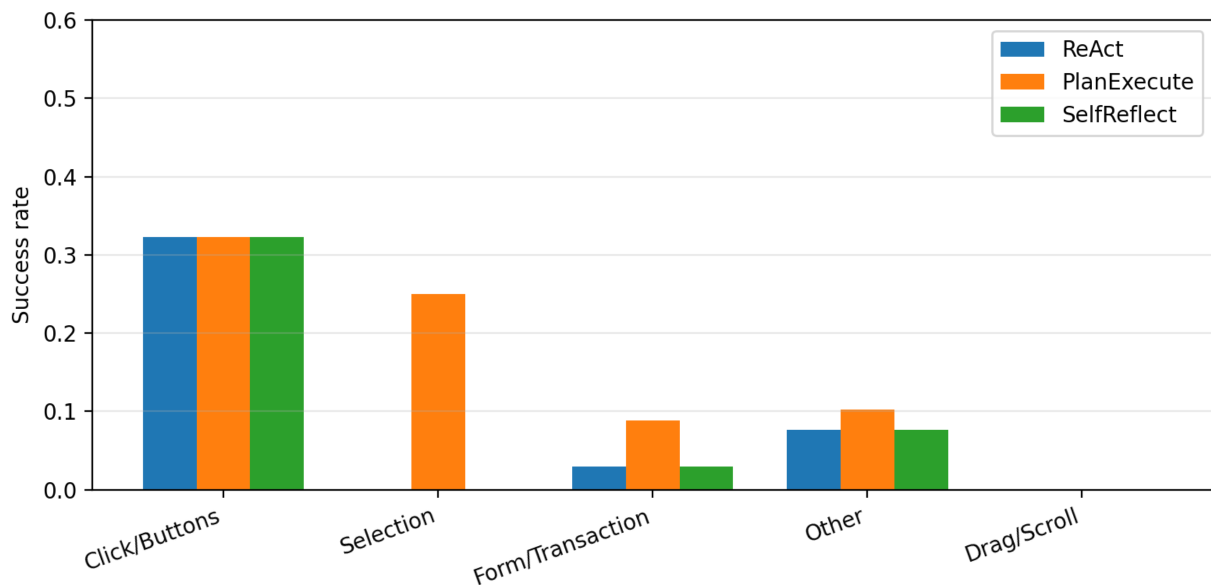


Figure 4. Success rate by coarse interaction category (grouped by agent)

These category differences are directly actionable for automated usability testing. A regression pipeline that relies on an agent like ReAct effectively checks only click-like affordances, while a planned policy broadens coverage to include some form-filling and selection controls. Conversely, interactions such as drag-and-drop, sliders, and scrolling remain outside the agent's competence and therefore require either explicit action primitives (e.g., drag trajectories) or a different observation model (e.g., pixel-based affordance detection).

Success rate versus difficulty proxy. Figure 2 plots success rate as a function of the DOM-derived difficulty proxy binned into five quantiles. Table 5 reports bin ranges and per-agent success. The proxy emphasizes surface interaction complexity rather than semantic reasoning, so it should be interpreted as "UI control complexity" rather than "task complexity." The observed curve is non-

monotonic: the lowest proxy bin contains tasks with very low counts of conventional clickables/inputs, which includes non-standard interaction widgets and therefore yields 0% success across all strategies. Mid-to-high bins contain many standard forms and button-heavy layouts where Plan-Execute's structured behavior helps, producing the highest bin-level success (34.8% in bin 4). The highest bin also includes dense interfaces that require precise element selection, reducing success to 11.5% for all three strategies.

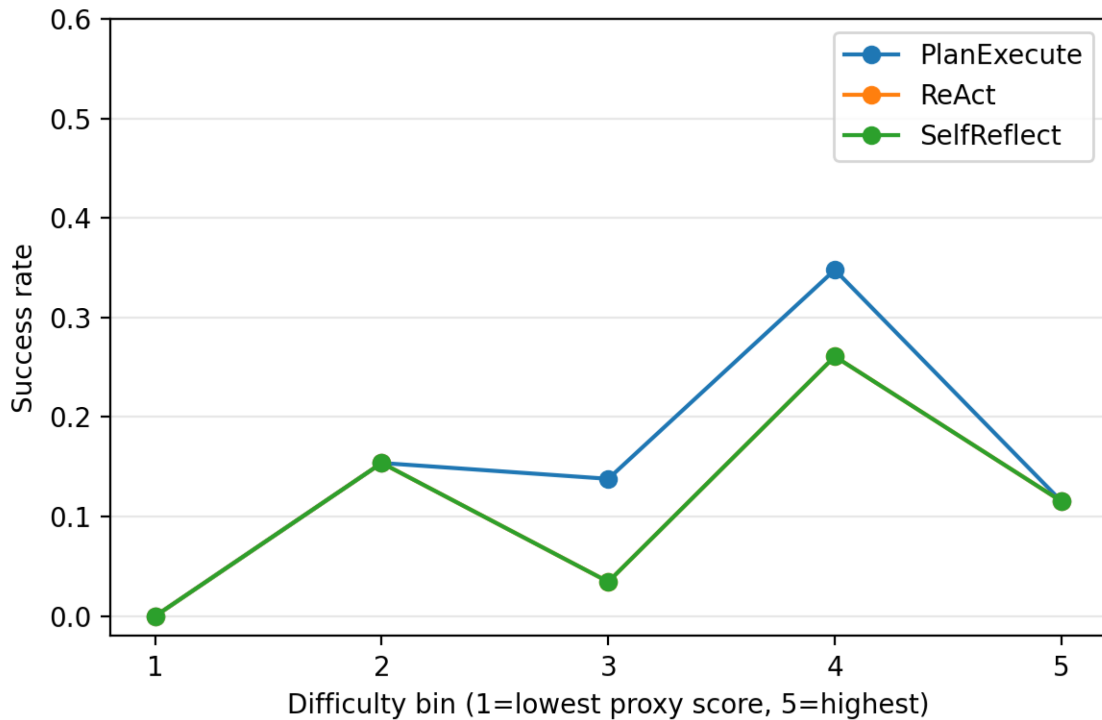


Figure 5. Success rate versus DOM-derived difficulty bins (five quantiles)

Table 5. Difficulty-bin ranges and success rates by agent (seed=0)

difficulty_bin	tasks	min_score	max_score	mean_score	ReAct_success_rate	PlanExecute_success_rate	SelfReflect_success_rate
1.0	26.0	0.6182	0.8439	0.758	0.0	0.0	0.0
2.0	26.0	0.8468	1.1405	1.028	0.1538	0.1538	0.1538
3.0	29.0	1.1437	1.3202	1.2293	0.0345	0.1379	0.0345
4.0	23.0	1.3231	1.467	1.397	0.2609	0.3478	0.2609
5.0	26.0	1.4695	2.4536	1.6824	0.1154	0.1154	0.1154

For usability testing, the primary value of the difficulty curve is comparative rather than absolute. The curve identifies a region where the agent's competence saturates: in this run, standard button-heavy interfaces are within reach, while non-standard widgets and dense pages requiring semantic disambiguation remain failure-dominated. This type of curve can be used as a "coverage dashboard" that tracks whether improvements in grounding or planning shift competence toward harder bins over time.

Efficiency patterns and step budgets. Figure 4 and Table 3 show that Plan-Execute uses fewer actions on average than ReAct and Self-Reflect (1.32 vs. 2.22 and 2.21 actions per task). This

is expected because Plan-Execute commits earlier to a structured fill/click/submit sequence, while ReAct and Self-Reflect more often continue searching for a matching target. The same structure lowers the frequency of step-limit failures (7 for Plan-Execute vs. 22 for ReAct and Self-Reflect) by steering the agent to terminating actions such as submit. In contrast, ReAct and Self-Reflect more often stall after an unmatched click target or after failing to identify a submit affordance.

Failure taxonomy and dominant bottlenecks. Table 6 and Figure 6 aggregate failures into a small taxonomy. For ReAct and Self-Reflect, the most frequent failure is WrongOutcome (51 tasks each), where the environment terminates with a negative reward after an incorrect action sequence. The second-largest failure contributors are ElementNotFound (24 tasks) and StepLimit (22 tasks). Plan-Execute shows the same ElementNotFound count (24) but a higher WrongOutcome count (62) and a lower StepLimit count (15). This pattern indicates that planning increases “decisiveness”: the agent reaches an end state more often, but the end state is frequently incorrect. In usability terms, WrongOutcome corresponds to errors where the agent misinterprets which control satisfies the instruction, while ElementNotFound corresponds to discoverability failures where the correct affordance is not located by the grounding mechanism.

Table 6. Outcome and failure-category distribution (count and % of tasks)

Failure/Outcome	ReAct	PlanExecute	SelfReflect
Success	14 (10.8%)	19 (14.6%)	14 (10.8%)
WrongOutcome	51 (39.2%)	62 (47.7%)	51 (39.2%)
ElementNotFound	24 (18.5%)	24 (18.5%)	24 (18.5%)
InputNotFound	0 (0.0%)	0 (0.0%)	0 (0.0%)
SubmitNotFound	2 (1.5%)	2 (1.5%)	2 (1.5%)
StepLimit	22 (16.9%)	7 (5.4%)	22 (16.9%)
NoAction	16 (12.3%)	16 (12.3%)	16 (12.3%)
TimeBudget	0 (0.0%)	0 (0.0%)	0 (0.0%)
SystemError	1 (0.8%)	0 (0.0%)	1 (0.8%)
Other	0 (0.0%)	0 (0.0%)	0 (0.0%)

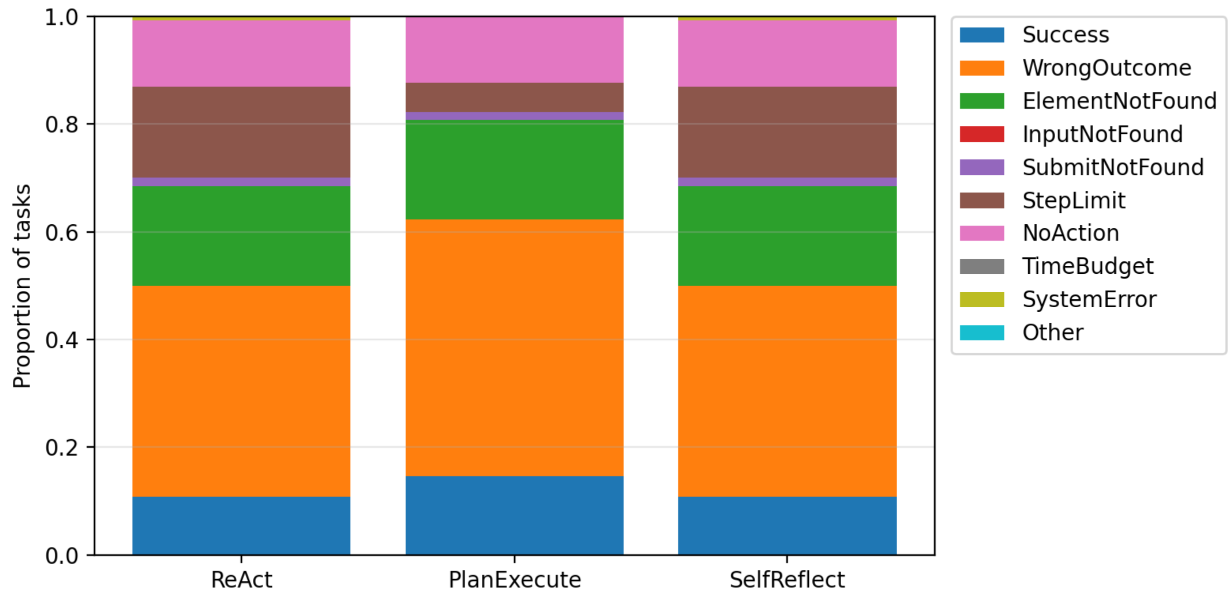


Figure 6. Stacked distribution of outcomes and failure categories by agent

Table 7 lists the top raw failure reasons. Across strategies, a small number of mechanisms dominate: negative_reward (incorrect termination), element_not_found:no_match (no clickable element matches the extracted target text), element_not_found:no_targets (instruction contains click language but yields no extracted targets), step_limit (no terminating action within 10 steps), and no_action (no rule triggers an action). The appearance of element_not_found:no_targets highlights a practical issue for instruction-driven agents: linguistic variability can prevent target extraction even when the interface contains the correct element. In an LLM-based agent, this corresponds to prompt sensitivity or insufficient instruction parsing; in our deterministic controller, it corresponds to the limitations of regex-based extraction.

Table 7. Top failure reasons per agent (raw strings, failures only)

Rank	ReAct reason	ReAct count	PlanExecute reason	PlanExecute count	SelfReflect reason	SelfReflect count
1	negative_reward	51	negative_reward	62	negative_reward	51
2	step_limit	22	element_not_found:no_match	18	step_limit	22
3	element_not_found:no_match	18	no_action	16	element_not_found:no_match	18
4	no_action	16	step_limit	7	no_action	16
5	element_not_found:no_targets	6	element_not_found:no_targets	6	element_not_found:no_targets	6
6	submit_not_found:no_submit	2	submit_not_found:no_submit	2	submit_not_found:no_submit	2

Per-task structure and agent complementarity. Figure 7 provides a per-task success heatmap sorted by difficulty proxy. The heatmap shows that success is sparse and clustered: the same subset of tasks is solved by all strategies, and Plan-Execute contributes a small number of additional successes. Because the solved task sets are nested (Plan-Execute ⊇ ReAct = Self-Reflect), the ensemble benefit of running multiple strategies is limited in this run. In a true LLM setting where stochastic sampling yields diverse trajectories, one would expect greater complementarity; the current deterministic instantiation instead reveals the structural advantage of explicit planning when the grounding primitive can execute the plan reliably.

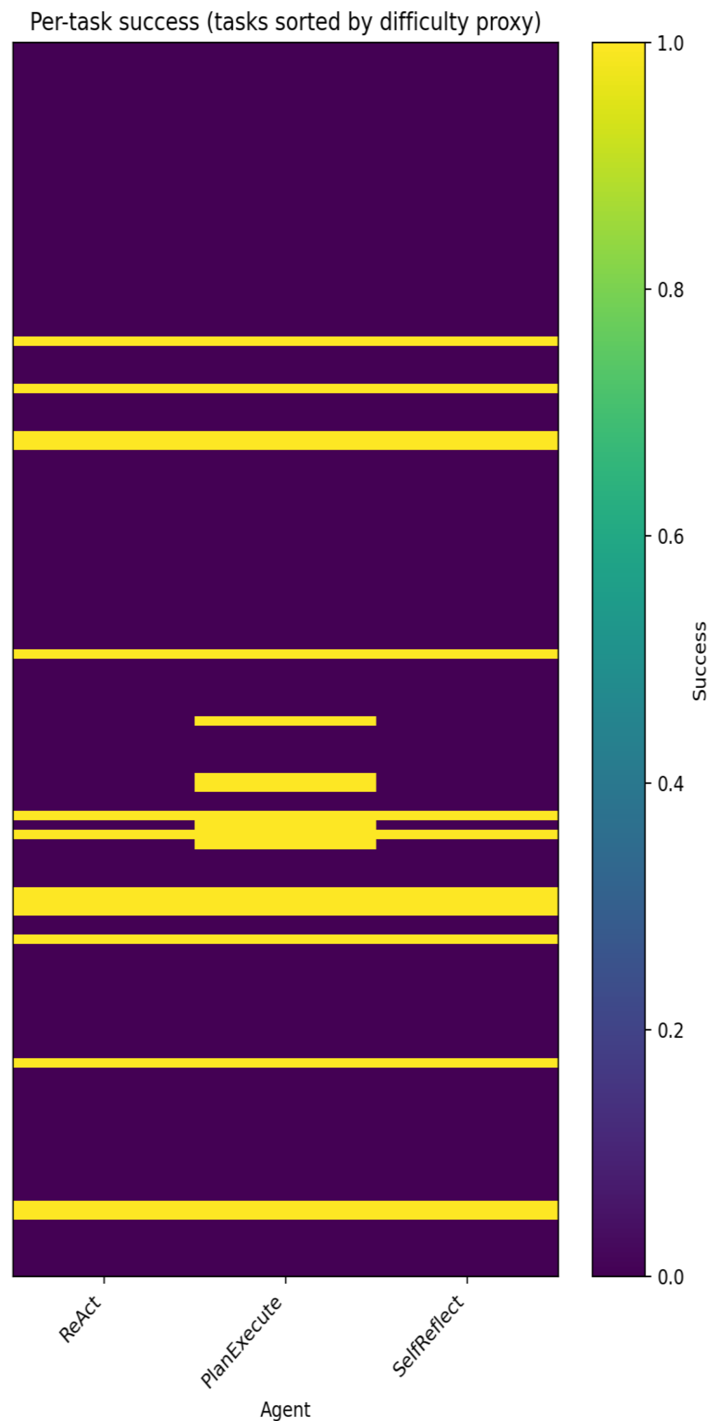


Figure 7. Per-task success heatmap sorted by the difficulty proxy (seed=0)

Implications for LLM usability agents. The results identify three concrete design requirements for scaling automated usability testing with LLM agents. (1) Semantic grounding beyond string matching: many failures arise because the agent cannot map a linguistic target to an element label or role. Approaches that combine DOM structure, accessibility trees, and language embeddings are expected to reduce ElementNotFound failures. (2) Plan validation and state checks: Plan-Execute increases WrongOutcome failures by committing to a plausible but incorrect plan. Adding intermediate validation—checking whether a filled field changed, whether an option was selected, or whether a confirmation dialog appeared—would convert some wrong-outcome terminations into recoverable detours. (3) Expanded action primitives for rich widgets: the 0% success rate on drag/scroll tasks indicates that action-space completeness is a hard requirement for coverage; without scroll and drag, large classes of UIs remain untestable regardless of reasoning quality.

Actionability for UI designers and engineers. The most useful output of an automated usability agent is a ranked list of friction points rather than a single aggregate score. The tables and figures in this paper suggest an operational workflow. First, treat task success as a gate for effectiveness and track it as a trend line; regressions are tasks that flip from success to failure under the same seed and harness. Second, use failure categories to route issues: ElementNotFound failures motivate label/affordance audits (e.g., ensure buttons have descriptive text and accessible labels), WrongOutcome failures motivate disambiguation (e.g., reduce duplicate button text, clarify call-to-action wording), and StepLimit/NoAction failures motivate flow instrumentation (e.g., ensure the next action is discoverable without hidden interactions). Third, analyze category slices: if a product UI shows systematic failures on selection widgets, prioritize improving widget accessibility and providing unambiguous option labels; if form/transaction flows dominate failures, prioritize consistent field labeling and inline validation messages that provide clear guidance. This mapping is consistent with classic usability principles such as match-to-language, recognition, and error prevention [2], but it is expressed as a concrete, automatable triage process. Finally, use per-task logs and screenshots to create reproducible bug reports: because the harness is deterministic, the same failure can be replayed locally, enabling rapid iteration on both the UI and the agent grounding logic.

Using agents as usability probes. Even with low absolute success, an agent can still support usability testing by producing consistent failure traces. For example, repeated element-not-found failures on a task indicate that the UI label is not easily matchable to the instruction phrasing, suggesting discoverability or wording issues. Wrong-outcome failures often indicate ambiguous control semantics (multiple similar buttons) or missing constraints in the instruction. In practice, a usability-testing pipeline can execute agent episodes nightly, track success and efficiency metrics over time, and attach failure logs and screenshots to bug reports. This reframes agent evaluation as a form of automated "user journey monitoring" for UI regressions, complementing classic script-based end-to-end tests.

Case study: tasks uniquely solved by Plan-Execute. The five tasks solved only by Plan-Execute are informative because they isolate the value of explicit planning under a shared grounding mechanism. choose-date-easy and choose-date-medium require selecting a date value and then completing the interaction; enter-text and enter-text-dynamic require reliable value entry; and use-autocomplete requires widget-mediated input through a suggestion mechanism. In these tasks, the Plan-Execute scaffold is more effective because it explicitly sequences value-setting, target

selection, and confirmation steps. The additional successes suggest that a plan-first policy is more reliable than a purely reactive policy when the task requires more than a single click and when the grounding primitive can identify the relevant control.

Table 8. Tasks solved only by Plan-Execute (seed=0)

Tasks solved only by PlanExecute (seed=0)
choose-date-easy
choose-date-medium
enter-text
enter-text-dynamic
use-autocomplete

Why Self-Reflect did not help in this run. Reflection-based agents are designed to recover from errors by analyzing failures and revising behavior [8]. In our deterministic instantiation, self-reflection is reduced to stagnation detection plus a generic fallback (submit, then click the first clickable element). Because most failures in MiniWoB++ arise from semantic grounding errors (choosing the wrong element) rather than from lack of an action, the fallback rarely redirects the agent toward the correct target. As a result, Self-Reflect produces the same success vector as ReAct and slightly higher mean time due to additional fallback actions. This outcome underscores a key principle for usability agents: reflection is only as effective as the alternative hypotheses the agent can generate. In a true LLM setting, reflection can generate new target candidates, reinterpret instructions, or propose different tool calls; in a fixed heuristic controller, fallback actions are largely blind.

Effectiveness–efficiency trade-offs. From an engineering standpoint, the success improvements of Plan-Execute are achieved with minimal cost: mean wall-clock time remains below 0.4 seconds per task on average, and the mean number of actions is just over one step. This indicates that adding planning structure does not introduce a prohibitive computational overhead in a browser-automation harness. At the same time, the higher WrongOutcome count for Plan-Execute (62) reveals a qualitative risk: a plan can confidently execute an incorrect sequence and thereby terminate with negative reward. In usability diagnostics, this distinction matters. A wrong outcome suggests ambiguous interface semantics or insufficient instruction grounding, whereas a step-limit failure suggests interaction dead-ends or missing affordances. A practical pipeline may therefore prefer an agent that fails early and interpretable (e.g., element-not-found) over one that fails late with a wrong outcome unless additional logging captures the intermediate action rationale.

Connecting failures to classic usability heuristics. Nielsen's heuristics emphasize match between system and real-world language, recognition rather than recall, and error prevention [2]. ElementNotFound failures align with mismatch and recognition failures: the interface labels and affordances do not align with the agent's phrasing model (or, in a human study, with users' mental models). WrongOutcome failures resemble error-prevention issues: the UI allows a plausible but incorrect action, and the agent lacks feedback to detect the error before committing. StepLimit and NoAction failures map to visibility-of-system-status issues: the agent cannot identify progress cues or next steps. While an automated agent does not experience "satisfaction" directly, these mappings show how quantitative agent traces can be translated into familiar usability categories for designers.

On the difficulty proxy and its use in practice. The DOM-derived difficulty proxy is intentionally simple to ensure reproducibility and low measurement overhead. It captures the idea

that pages with many controls require more disambiguation and that longer instructions may require more structured interpretation. The proxy also has clear failure cases: tasks dominated by canvas-like widgets or custom interaction surfaces can appear "easy" because they expose few standard inputs or buttons, yet they are difficult for our action model. In a production usability-testing pipeline, the proxy can be refined by incorporating additional observable signals such as the depth of the DOM tree, the number of unique text labels, the entropy of clickable texts, or the presence of scrollable containers. The key requirement is that the proxy remains stable across runs so that success–difficulty curves remain comparable over time.

From benchmark outcomes to actionable UI reports. To be useful for product teams, the raw per-task success matrix (Table 10) should be translated into a reporting layer. A simple approach is to attach to each failed episode a structured record: the instruction, the list of extracted targets and key–value pairs, the sequence of executed actions with timestamps, and a final screenshot. Aggregating these records across nightly runs allows engineers to identify regressions (tasks that previously succeeded now fail), while aggregating by failure category highlights systematic usability risks (e.g., frequent element-not-found failures after a label change). The MiniWoB++ benchmark does not model user satisfaction directly, but the same reporting infrastructure can be extended to include proxy satisfaction metrics such as the number of backtracks, repeated clicks, or time spent without progress, which often correlate with frustration in human studies.

Relation to broader web-agent benchmarks. While this study focuses on MiniWoB++, recent benchmarks evaluate agents on more realistic websites and multi-page workflows, often requiring tool use and long-horizon planning [14]–[17]. MiniWoB++ remains valuable for usability-agent research because it isolates the interaction layer: a failure can be traced to a specific UI element and instruction without external content variability. The agent strategies evaluated here can be reused in broader benchmarks by replacing the grounding layer with an accessibility-tree or vision-language model and by extending the action set to include navigation, scrolling, and multi-tab control. The diagnostic methodology—success metrics, efficiency metrics, and failure taxonomies—transfers directly.

Design recommendations for integrating LLM agents. The empirical results identify concrete engineering priorities. First, the grounding layer should unify multiple evidence sources: visible text, aria labels, structural cues (e.g., "label for input"), and the accessibility tree. Second, planning should be constrained by verifiable subgoals: after a fill action, check that the field contains the intended value; after a click, check that the expected state change occurred (e.g., a modal opened or a selection highlight changed). Third, the agent should support targeted exploration under uncertainty, such as enumerating candidate elements and trying them in a ranked order, rather than resorting to blind fallbacks. Finally, the logging layer should be treated as a first-class artifact; for usability testing, the explanation of failure is often more valuable than the raw success rate.

Failure patterns by interaction category. Category-level views clarify whether failures are due to missing action primitives or due to grounding ambiguity within the existing action set. Drag/scroll tasks fail uniformly because the agent does not execute drag trajectories or scrolling, producing either step-limit or wrong-outcome terminations depending on the task controller. Selection tasks fail for ReAct and Self-Reflect primarily because target extraction yields no usable candidate or because the selected element does not correspond to the intended option. Plan-Execute converts some of these into successes by explicitly pairing a selection step with a terminating submit step. Form/transaction tasks show a mixture of failure causes: element-not-found failures occur

when field labels cannot be matched to inputs, while wrong-outcome failures occur when the agent fills an incorrect field or submits prematurely. This breakdown indicates that improving the mapping between instruction semantics and form fields is a high-leverage path to higher automated usability coverage.

Budget effects and practical tuning. The 10-step limit and the 2-second time budget define an operational regime that resembles an automated regression check: the agent is expected to succeed quickly or fail with a diagnostic. Within this regime, step-limit failures are interpreted as "non-convergent interaction," which is a useful usability signal because real users often abandon flows that do not make progress. Increasing the step limit would raise success on some tasks by allowing more exploration, but it would also reduce the interpretability of failures and increase evaluation cost. A practical system therefore exposes the budgets as tunable parameters tied to product needs: low budgets for nightly regression checks and higher budgets for deeper exploratory audits. In either setting, the same logging and taxonomy infrastructure supports interpretation of outcomes.

The scope and external-validity constraints of these findings are addressed in the dedicated Limitations section. The key point for interpreting the results is that the deterministic setup is intended as a reproducible baseline and diagnostic tool rather than as a claim about the maximum capability of fully learned LLM agents.

Statistical comparison. Table 9 reports exact McNemar tests for paired task success. Plan-Execute solves 5 tasks that ReAct fails, while ReAct solves 0 tasks that Plan-Execute fails, yielding an exact p-value of 0.0625. The same discordance holds for Plan-Execute versus Self-Reflect. ReAct and Self-Reflect have identical success vectors (0 discordant tasks), confirming that the implemented reflection mechanism does not change effectiveness under the present action space and heuristics. The paired-test result supports the conclusion that the observed advantage of explicit planning is robust at the task level for this deterministic run.

Table 9. Paired exact McNemar tests on per-task success (seed=0)

Comparison	A=1,B=0 (n10)	A=0,B=1 (n01)	Exact McNemar p-value
PlanExecute vs ReAct	5	0	0.0625
PlanExecute vs SelfReflect	5	0	0.0625
ReAct vs SelfReflect	0	0	1.0

Table 10. Per-task summary: category, difficulty proxy, and success by agent (seed=0)

task	category	difficulty_score	ReAct	PlanExecute	SelfReflect
click-shape	Click/Buttons	0.6182	0	0	0
click-menu	Click/Buttons	0.6356	0	0	0
click-color	Click/Buttons	0.6516	0	0	0
click-link	Click/Buttons	0.6664	0	0	0
drag-items	Drag/Scroll	0.6931	0	0	0
hot-cold	Other	0.6931	0	0	0

task	category	difficulty_score	ReAct	PlanExecute	SelfReflect
drag-items-grid	Drag/Scroll	0.6993	0	0	0
grid-coordinate	Other	0.7222	0	0	0
email-inbox-forward-nl	Form/Transaction	0.7427	0	0	0
ascending-numbers	Other	0.7427	0	0	0
moving-items	Other	0.7475	0	0	0
tic-tac-toe	Other	0.7522	0	0	0
social-media	Other	0.7657	0	0	0
hover-shape	Other	0.7742	0	0	0
email-inbox-nl-turk	Form/Transaction	0.7742	0	0	0
chase-circle	Other	0.7978	0	0	0
email-inbox-forward	Form/Transaction	0.8015	0	0	0
email-inbox-forward-nl-turk	Form/Transaction	0.8086	0	0	0
email-inbox-noscroll	Drag/Scroll	0.8121	0	0	0
daily-calendar	Other	0.8189	0	0	0
click-pie	Click/Buttons	0.8222	0	0	0
click-pie-nodelay	Click/Buttons	0.8222	0	0	0
email-inbox-delete	Form/Transaction	0.8254	0	0	0
email-inbox	Form/Transaction	0.8379	0	0	0
email-inbox-reply	Form/Transaction	0.8379	0	0	0
click-collapsible-2-nodelay	Click/Buttons	0.8439	0	0	0
click-collapsible-2	Click/Buttons	0.8468	0	0	0
terminal	Other	0.8553	0	0	0

task	category	difficulty_score	ReAct	PlanExecute	SelfReflect
email-inbox-star-reply	Form/Transaction	0.8608	0	0	0
email-inbox-important	Form/Transaction	0.8608	0	0	0
focus-text	Form/Transaction	0.8782	0	0	0
click-test	Click/Buttons	0.89	1	1	1
navigate-tree	Other	0.8932	0	0	0
drag-single-shape	Drag/Scroll	1.0286	0	0	0
drag-shapes	Drag/Scroll	1.0497	0	0	0
drag-circle	Drag/Scroll	1.0546	0	0	0
click-dialog	Click/Buttons	1.0595	1	1	1
click-collapsible	Click/Buttons	1.0642	0	0	0
click-collapsible-nodelay	Click/Buttons	1.0642	0	0	0
use-slider	Drag/Scroll	1.0642	0	0	0
enter-time	Form/Transaction	1.0688	0	0	0
click-test-2	Click/Buttons	1.0724	1	1	1
click-test-transfer	Click/Buttons	1.0724	1	1	1
enter-date	Form/Transaction	1.0732	0	0	0
click-shades	Click/Buttons	1.0819	0	0	0
stock-market	Other	1.0943	0	0	0
use-slider-2	Drag/Scroll	1.1205	0	0	0
highlight-text-2	Form/Transaction	1.124	0	0	0
highlight-text	Form/Transaction	1.1308	0	0	0
right-angle	Other	1.1373	0	0	0
click-tab-2-easy	Click/Buttons	1.1405	0	0	0
circle-center	Form/Transaction	1.1405	0	0	0
enter-password	Form/Transaction	1.1437	0	0	0
draw-line	Drag/Scroll	1.1529	0	0	0

task	category	difficulty_score	ReAct	PlanExecute	SelfReflect
social-media-some	Other	1.1529	0	0	0
order-food	Form/Transaction	1.1529	0	0	0
find-greatest	Other	1.1558	0	0	0
drag-box	Drag/Scroll	1.1587	0	0	0
drag-cube	Drag/Scroll	1.1587	0	0	0
bisect-angle	Other	1.1645	0	0	0
social-media-all	Other	1.17	0	0	0
click-menu-2	Click/Buttons	1.1781	0	0	0
drag-shapes-2	Drag/Scroll	1.1833	0	0	0
focus-text-2	Form/Transaction	1.1905	0	0	0
click-tab	Click/Buttons	1.1905	1	1	1
find-midpoint	Other	1.1981	0	0	0
drag-sort-numbers	Drag/Scroll	1.2206	0	0	0
draw-circle	Drag/Scroll	1.2206	0	0	0
guess-number	Other	1.2248	0	0	0
click-button-sequence	Click/Buttons	1.2371	0	0	0
generate-number	Other	1.2768	0	0	0
use-autocomplete	Other	1.282	0	1	0
button-delay	Click/Buttons	1.303	0	0	0
multi-orderings	Form/Transaction	1.3074	0	0	0
choose-list	Selection	1.3114	0	0	0
click-tab-2-medium	Click/Buttons	1.3166	0	0	0
click-option	Click/Buttons	1.317	0	0	0
choose-date-easy	Selection	1.3202	0	1	0
choose-date-medium	Selection	1.3202	0	1	0

task	category	difficulty_score	ReAct	PlanExecute	SelfReflect
choose-date-nodelay	Selection	1.3202	0	0	0
choose-date	Selection	1.3202	0	0	0
click-checkboxes	Click/Buttons	1.3231	1	1	1
enter-text-dynamic	Form/Transaction	1.3369	0	1	0
use-colorwheel	Other	1.3409	1	1	1
enter-text	Form/Transaction	1.3448	0	1	0
click-scroll-list	Drag/Scroll	1.3523	0	0	0
scroll-text-2	Drag/Scroll	1.356	0	0	0
use-autocomplete-nodelay	Other	1.3596	0	0	0
text-transform	Form/Transaction	1.3666	0	0	0
unicode-test	Other	1.3759	1	1	1
use-colorwheel-2	Other	1.3894	1	1	1
click-checkboxes-transfer	Click/Buttons	1.3907	1	1	1
visual-addition	Other	1.3955	0	0	0
read-table	Selection	1.3984	0	0	0
click-dialog-2	Click/Buttons	1.4022	1	1	1
simple-algebra	Other	1.418	0	0	0
enter-text-2	Form/Transaction	1.4207	0	0	0
find-word	Other	1.4383	0	0	0
simple-arithmetic	Other	1.4454	0	0	0
login-user-popup	Form/Transaction	1.45	0	0	0
login-user	Form/Transaction	1.45	0	0	0
resize-textarea	Drag/Scroll	1.4545	0	0	0
click-tab-2	Click/Buttons	1.4556	0	0	0
simon-says	Other	1.467	0	0	0

task	category	difficulty_score	ReAct	PlanExecute	SelfReflect
search-engine	Other	1.4695	0	0	0
buy-ticket	Form/Transaction	1.4765	0	0	0
count-shape	Other	1.4798	0	0	0
sign-agreement	Form/Transaction	1.5227	1	1	1
click-widget	Click/Buttons	1.5381	0	0	0
click-tab-2-hard	Click/Buttons	1.556	0	0	0
form-sequence	Form/Transaction	1.5681	0	0	0
copy-paste	Other	1.5753	0	0	0
scroll-text	Drag/Scroll	1.5778	0	0	0
read-table-2	Selection	1.585	0	0	0
text-editor	Form/Transaction	1.5956	0	0	0
identify-shape	Other	1.6078	0	0	0
multi-layouts	Other	1.616	0	0	0
use-spinner	Other	1.6187	0	0	0
phone-book	Form/Transaction	1.6317	0	0	0
book-flight	Form/Transaction	1.6351	0	0	0
count-sides	Other	1.656	0	0	0
book-flight-nodelay	Form/Transaction	1.676	0	0	0
click-checkboxes-soft	Click/Buttons	1.6978	1	1	1
click-button	Click/Buttons	1.7604	1	1	1
odd-or-even	Other	1.7737	0	0	0
copy-paste-2	Other	1.7881	0	0	0
form-sequence-3	Form/Transaction	1.858	0	0	0
click-checkboxes-large	Click/Buttons	1.9531	0	0	0
form-sequence-2	Form/Transaction	2.0729	0	0	0
number-checkboxes	Selection	2.4536	0	0	0

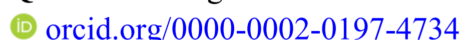
CONCLUSION

This study demonstrates that LLM-agent-style automated usability testing can provide reproducible and diagnostically meaningful evidence for evaluating web-interface interaction, particularly when benchmark outcomes are interpreted beyond aggregate success scores. Through a controlled evaluation of 130 MiniWoB++ tasks and 390 total episodes, the findings show that Plan-Execute outperformed ReAct and Self-Reflect in overall success, especially on form/transaction and selection tasks, indicating that explicit planning can improve task coverage when interaction goals require ordered value entry, selection, and confirmation. However, the consistently low absolute success rates and the complete failure on drag/scroll tasks reveal that current deterministic agent scaffolds remain strongly constrained by limited action primitives and imperfect element grounding. The dominance of WrongOutcome and ElementNotFound failures further indicates that the main bottlenecks are not merely computational efficiency, but semantic alignment between task instructions, interface labels, accessible elements, and verifiable intermediate states. These findings contribute a reproducible baseline and a usability-oriented failure taxonomy that can help researchers and practitioners transform agent traces into actionable UI diagnostics. Future automated usability-testing frameworks should therefore prioritize stronger semantic grounding, plan validation, richer interaction primitives, accessibility-aware observation layers, and designer-facing reports that translate agent failures into concrete improvements for interface clarity, discoverability, and interaction robustness.

LIMITATIONS

This study has several limitations that should be considered when interpreting the findings. First, the evaluation was conducted in the controlled MiniWoB++ environment, which is useful for isolating interaction failures but does not fully represent the complexity, visual richness, dynamic content, and multi-page navigation patterns of real-world websites. Second, each task was executed only once under a fixed seed, so the results should be understood as a reproducible baseline rather than a comprehensive estimate of performance across multiple randomized runs or stochastic agent behaviors. Third, the agents were implemented through deterministic DOM-grounded policies rather than fully generative LLM agents; therefore, the findings primarily reflect the effect of policy scaffolds and grounding heuristics, not the maximum capability of contemporary large language models. Fourth, the restricted action space—limited mainly to clicking, typing or setting values, and submit heuristics—constrained performance on interaction types requiring scrolling, dragging, visual perception, or complex widget manipulation. Fifth, the study focused on effectiveness, efficiency, and failure diagnostics, while subjective usability dimensions such as user satisfaction, perceived workload, frustration, and trust could not be directly measured because the evaluation used synthetic agents rather than human participants. Finally, the DOM-derived difficulty proxy provides a reproducible indicator of interface complexity, but it may underestimate tasks involving custom widgets, hidden interactions, or semantic ambiguity. Future studies should extend the evaluation to multiple seeds, richer action primitives, accessibility-tree and vision-based grounding, real-world web environments, and hybrid comparisons with human usability testing to strengthen external validity and practical applicability.

AUTHOR INFORMATION

Corresponding Author
Qi Xin – Management Information Systems, University of Pittsburgh (United States);

orcid.org/0000-0002-0197-4734
Email: qix29@pitt.edu

CONFLICT OF INTEREST

The authors declare no conflict of interest.

DECLARATION OF USE OF AI IN SCIENTIFIC WRITING

The authors used ChatGPT for word refinement during the preparation of this work. After utilizing the tool, the authors thoroughly reviewed and edited the content as necessary, assuming full responsibility for the publication's content.

REFERENCES

[1] International Organization for Standardization, "ISO 9241-11:2018 Ergonomics of human-system interaction—Part 11: Usability: Definitions and concepts," Geneva, Switzerland: International Organization for Standardization, 2018.
[2] J. Nielsen, *Usability Engineering*. San Diego, CA, USA: Academic Press, 1993.
[3] J. Brooke, "SUS: A 'quick and dirty' usability scale," in *Usability Evaluation in Industry*, P. W. Jordan, B. Thomas, B. A. Weerdmeester, and I. L. McClelland, Eds. London, U.K.: Taylor & Francis, 1996, pp. 189–194.
[4] E. Z. Liu, K. Guu, P. Pasupat, T. Shi, and P. Liang, "Reinforcement learning on web interfaces using workflow-guided exploration," in *Proc. International Conference on Learning Representations (ICLR)*, 2018. [Online]. Available: https://arxiv.org/abs/1802.08802
[5] T. Shi, A. Karpathy, L. Fan, J. Hernandez, and P. Liang, "World of Bits: An open-domain platform for web-based agents," in *Proc. 34th International Conference on Machine Learning (ICML)*, *Proceedings of Machine Learning Research*, vol. 70, 2017, pp. 3135–3144. [Online]. Available: https://proceedings.mlr.press/v70/shi17a.html
[6] Farama Foundation, "miniwob-plusplus: MiniWoB++ environment for Gymnasium," GitHub repository, 2023. [Online]. Available: https://github.com/Farama-Foundation/miniwob-plusplus
[7] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proc. International Conference on Learning Representations (ICLR)*, 2023. [Online]. Available: https://arxiv.org/abs/2210.03629
[8] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. [Online]. Available: https://arxiv.org/abs/2303.11366
[9] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K.-W. Lee, and E.-P. Lim, "Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models," in *Proc. 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023, pp. 2609–2634. https://doi.org/10.18653/v1/2023.acl-long.147
[10] B. Xu, Z. Peng, B. Lei, S. Mukherjee, Y. Liu, and D. Xu, "ReWOO: Decoupling reasoning from observations for efficient augmented language models," arXiv:2305.18323, 2023. [Online]. Available: https://arxiv.org/abs/2305.18323
[11] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of

Thoughts: Deliberate problem solving with large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. [Online]. Available: https://arxiv.org/abs/2305.10601

[12] J. Wei *et al*., "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022, pp. 24824–24837.

[13] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. [Online]. Available: https://arxiv.org/abs/2302.04761

[14] S. Yao *et al*., "WebShop: Towards scalable real-world web interaction with grounded language agents," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. [Online]. Available: https://arxiv.org/abs/2207.01206

[15] S. Zhou *et al*., "WebArena: A realistic web environment for building autonomous agents," in *Proc. International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://arxiv.org/abs/2307.13854

[16] T. Le Sellier De Chezelles *et al*., "The BrowserGym ecosystem for web agent research," arXiv:2412.05467, 2024. [Online]. Available: https://arxiv.org/abs/2412.05467

[17] Q. Wu *et al*., "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," arXiv:2308.08155, 2023. [Online]. Available: https://arxiv.org/abs/2308.08155

[18] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. [Online]. Available: https://arxiv.org/abs/2405.15793

[19] L. Ouyang *et al*., "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022, pp. 27730–27744. [Online]. Available: https://arxiv.org/abs/2203.02155

[20] P. F. Christiano *et al*., "Deep reinforcement learning from human preferences," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017. [Online]. Available: https://papers.nips.cc/paper/7017-deep-reinforcement-learning-from-human-preferences

[21] R. Nakano *et al*., "WebGPT: Browser-assisted question-answering with human feedback," arXiv:2112.09332, 2021. [Online]. Available: https://arxiv.org/abs/2112.09332

[22] J. Y. Koh *et al*., "VisualWebArena: Evaluating multimodal agents on realistic visual web tasks," in *Proc. 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. https://doi.org/10.18653/v1/2024.acl-long.50