# WEBCHALLENGER: A RELIABLE AND EFFICIENT GENERALIST WEB AGENT

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Browsing the web is effortless for people yet hard for language-model agents, and the strongest web agents depend on proprietary reasoning models too costly for the repetitive work they would be most useful for. We argue that the bottleneck is not the model but how the agent harness presents the web to it: a person attends only to the relevant part of a page, remembers how familiar websites are laid out, and performs routines such as filling a form as one fluent action, whereas an agent receives each page whole and acts one click at a time. We introduce WebChallenger, a harness that recovers each advantage through architecture rather than model scale. Its core is PageMem, a page representation built by fixed rules from the page's Document Object Model (DOM) that presents a page as a hierarchy of semantic sections, each with a one-sentence summary; on it we build an observation pipeline that reads only the relevant sections, a memory filled by a one-time offline crawl of each website, and workflows that execute multi-step interactions as single actions. With off-the-shelf open-weight models and no fine-tuning, WebChallenger sets the state of the art among small open-weight agents on WebArena, VisualWebArena, Online-Mind2Web, and WorkArena; a small base model run inside it outperforms the same model fine-tuned for web navigation in most published cases; and at a fixed backbone it outperforms the standard BrowserGym harness by 39 points and the top-ranked WebArena harness by 13–14 points on WebArena-lite. Carried over to frontier vision-language models, the same page decomposition (WebChallenger+) matches the task success of the harnesses these models ship with, Claude Code and Codex, on long-horizon enterprise tasks at 1.4–4.4$\times$ lower inference cost on the same backbone.

## 1 INTRODUCTION

Autonomous web navigation has long been a goal of AI research (Doorenbos et al., 1997): the web is one of the largest interactive environments in existence, and an agent that can operate it could automate much of the repetitive knowledge work that fills a working day. Large language models (LLMs) and vision-language models (VLMs) have made *web agents*, programs that read a web page and choose browser actions to complete a natural-language task, a practical research target (Marino & Marasović, 2025), yet even the strongest remain below human performance on realistic long-horizon tasks (Jang et al., 2026; Miyai et al., 2025), and the best depend on proprietary reasoning models whose cost per task rules out the repetitive work they would be most useful for.

This gap echoes Moravec's paradox (Moravec, 1988; Su, 2025): browsing is effortless for people yet difficult for models that excel at mathematics and code. We argue that the difficulty lies less in what the model knows about the web than in how the *agent harness*, the software around the model that turns a page into a prompt and a model output into a browser action, presents the web to it. Today's harnesses hand the model either a screenshot or the page's *accessibility tree*, a textual rendering of the browser's Document Object Model (DOM), as one flat input, and ask for one atomic action at a time. Compared with a person, such an agent lacks three things. *Selective attention*: a person focuses on the relevant region of a page and ignores the rest (Putkonen et al., 2023), whereas the agent ingests the entire page, and the relevant information is diluted among boilerplate. *Persistent memory*: a person remembers the layout and functionality of websites they have used before, whereas the agent starts every session with no knowledge of the site. *Procedural fluency*: a person executes



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

routines such as searching, choosing from a dropdown, or filling a form as one cohesive sequence, whereas the agent must re-observe the full page and re-reason for every click and keystroke.

We show that all three advantages can be recovered through agent architecture rather than model scale or training. The key is a shared abstraction that every mechanism reasons over, so that each generalizes across websites without site-specific code: **PageMem**, a page representation whose structure is constructed from the DOM by fixed rules and whose one-sentence section summaries are model-written. PageMem exposes a page as a hierarchy of semantic sections, such as a navigation bar, a product list, or a review form, so the agent can skim a page like a table of contents, expand only the sections that matter, and dispatch each section to a workflow by its type. On this substrate we build three mechanisms, one per missing advantage (Figure 1).

**Divide-and-conquer observation** reads a page in stages: the agent skims the section summaries, selects the relevant sections, extracts task-relevant details from each in a separate call, and condenses them into a one-paragraph observation, with long lists filtered chunk by chunk beforehand, so the input to every model call is bounded by one section or one 25-item chunk rather than by the page (Appendix E.2). **Exploration and memory** crawls each website once, offline and by fixed rules, recording the PageMem of every page, where each element leads, and what it reveals when clicked; at inference this supplies bookmarks, cached section summaries, and the contents of dropdown menus before they are opened. **Compound-action workflows** turn interactions whose intermediate steps change only part of the page, such as searching, choosing from a dropdown, or filling a form, into single agent actions, while navigations to a new page remain individual decisions; because workflows dispatch on section type they need no per-site code.

Using an off-the-shelf 32B LLM and a small vision model, without fine-tuning, WebChallenger achieves the best results among small open-weight agents on WebArena (Zhou et al., 2024), VisualWebArena (Koh et al., 2024), Online-Mind2Web (Xue et al., 2025), and WorkArena (Drouin et al., 2024), and a small base model run inside WebChallenger without fine-tuning outperforms the same model fine-tuned for web navigation in most published cases, as well as several agents backed by larger or proprietary models. The gains come from the harness: with the backbone model held fixed, WebChallenger improves WebArena-lite success by 39 points over BrowserGym's GenericAgent and by 13–14 points over WebTactix, the top-ranked WebArena agent, and ablations attribute the improvement to all three mechanisms, with the observation pipeline contributing most. Finally, the same page decomposition carried over to frontier VLMs (WebChallenger+) matches the success of Claude Code and Codex, the harnesses those models ship with, on long-horizon enterprise tasks at 1.4–4.4$\times$ lower inference cost on the same backbone.

## 2 Related Work

**Web Agents and Benchmarks.** LLM and vision-language model (VLM) agents map a task and the current page to browser actions, and are measured on execution-based benchmarks spanning simulated websites (Zhou et al., 2024; Koh et al., 2024), live websites (Xue et al., 2025), and enterprise software (Drouin et al., 2024). The strongest results come from proprietary computer-use agents (OpenAI, 2025; Marreed et al., 2025; Marino & Marasović, 2025) and frontier-model systems with learned tools or multi-agent planning (Prabhu et al., 2025; Zhang et al., 2024). Small open-weight agents instead close the gap by fine-tuning on synthesized, distilled, or production trajectories (Lai et al., 2024; Xu et al., 2025; Gandhi & Neubig, 2025; Shen et al., 2025; Awadallah et al., 2025; Xu et al., 2026; Lù & Reddy, 2026; Shen et al., 2024; Zhang et al., 2025b). A third line changes only the scaffold around a fixed model: research harnesses such as BrowserGym's GenericAgent (Chezelles et al., 2025), AgentOccam (Yang et al., 2025), and WebTactix (WebTactix), deployed open-source harnesses such as Browser Use (Müller & Žunic., 2024), Stagehand (Browserbase, 2024), and Skyvern (Skyvern AI, 2024), the computer-use agents that model vendors ship (OpenAI, 2025; Anthropic, 2024; Google DeepMind, 2025), and the coding agents Claude Code and Codex, which drive a browser through the same tool interface. WebChallenger is such a harness; we compare against GenericAgent and WebTactix at a fixed small backbone and against Claude Code and Codex at a fixed frontier backbone.

**Observation Refinement.** Web agents observe pages as text (Gur et al., 2018; Li et al., 2023; Kim et al., 2023), screenshots (Shaw et al., 2023; Hong et al., 2024; Gou et al., 2025; Pahuja et al., 2025; He et al., 2024; Zheng et al., 2024; Verma et al., 2024), or both (Furuta et al., 2024), all token-heavy and information-sparse, so many methods refine what the model reads. Text-based agents prune irrelevant

nodes from the HTML or from the accessibility tree, the browser's semantic view of the DOM built for screen readers (Gur et al., 2024; Deng et al., 2023; Kil et al., 2024; Lù et al., 2024; Lee et al., 2025; Abuelsaad et al., 2024; Kerboua et al., 2025), while vision-based agents focus attention on specific screen regions (Sarch et al., 2025b; Singh et al., 2025; Luo et al., 2025; Park et al., 2025). Feuillade--Montixi (2026) is closest to our approach, compressing HTML into an unfoldable representation that the agent expands on demand. Beyond web agents, long-context methods decompose a document into focused sub-prompts over chunks (Zhang et al., 2026; Chen et al., 2023; Jayalath et al., 2025; Lee et al., 2024), echoing the selective attention that distinguishes human reading from transformer attention (Zhao et al., 2024). WebChallenger applies this decomposition to a web agent, splitting pages along DOM structure, which preserves authored semantic grouping better than pixel-space cropping, so every prompt stays small regardless of page size.

**Agent Memory and Exploration.** Most web-agent memories are distilled from task trajectories as reusable workflows (Wang et al., 2024), reasoning insights and guidelines (Ouyang et al., 2025; Fu et al., 2024; Nekoei et al., 2025; Liu et al., 2025; Pang et al., 2025; Cheng et al., 2025), instruction manuals (Chen et al., 2024), or programs of thought (Sarch et al., 2025a). Exploration-driven methods instead synthesize training data by interacting with websites (Pahuja et al., 2025; Gandhi & Neubig, 2025; Su et al., 2025), or explore during execution via tree search and model-based planning (Koh et al., 2025; Gu et al., 2025). Our memory takes a complementary route: a deterministic crawl builds a site map of bookmarks, cached section summaries, and dropdown contents offline, requiring no task experience, demonstrations, or documentation and adding only a few tokens per prompt.

**Web Action Space.** Atomic click-and-type actions force re-observation after every micro-interaction, so several works raise the abstraction level by calling website APIs (Song et al., 2025) or inducing programmatic skills (Wang et al., 2025; Zheng et al., 2025a; He et al., 2025; Prabhu et al., 2025; Yu et al., 2025; Wang et al., 2026; Zhong et al., 2026). These approaches typically learn site-specific code, whereas our compound workflows operate over PageMem's abstract elements and sections and generalize across sites with no per-site adaptation.

# 3 METHOD

## 3.1 PROBLEM FORMULATION AND OVERVIEW

We frame web navigation as a sequential decision process in which an agent interacts with a web browser to complete a natural-language task. A task is a tuple $\tau = (I, u_0)$ consisting of an instruction $I$ and a starting URL $u_0$, which determines the initial website $w_0$. At each timestep $t$, the agent receives an observation $o_t$, maintains a compact history $h_t$ of prior interactions, and selects an action $a_t$ from a candidate set $\mathcal{A}_t$. A standard agent implements this loop as one model call $a_t = \pi(o_t, h_t)$ over a full accessibility tree or screenshot; WebChallenger keeps the loop but replaces $o_t$ with a summary $\hat{o}_t$ built from PageMem and lets one action execute several browser operations (Figure 1).

Given $\tau$, the agent loads the WebsiteMem $\mathcal{M}_{w_0}$, a URL-indexed collection of PageMems built by offline exploration (§3.3), and may select a bookmark set $B_\tau \subseteq \mathcal{M}_{w_0}$ of pages to keep available as navigation targets. At each timestep it (i) retrieves or constructs the PageMem $p_t$ of the current page, an ordered list of summarized sections and their interactive elements (§3.2); (ii) selects the relevant sections $S_t$ and condenses them into a task-focused observation $\hat{o}_t$ (§3.4); and (iii) chooses an action $a_t$ from a candidate set assembled from $S_t$, $B_\tau$, and visited URLs, executed either as one browser operation or as a workflow of several operations and LLM sub-calls (§3.5). The loop ends when the agent selects end-task and a verification call confirms completion, or after 30 steps (full procedure in Appendix A.4).

## 3.2 PAGEMEM

PageMem is an abstract page representation whose structure is constructed from the DOM by fixed rules and whose summaries are model-written; it serves as the common interface shared by the exploration (§3.3), observation (§3.4), and action (§3.5) components. It exposes a semantic, chunked view of a page while keeping the CSS selectors needed to drive the browser, so higher-level components operate on abstract sections and elements without site-specific code.

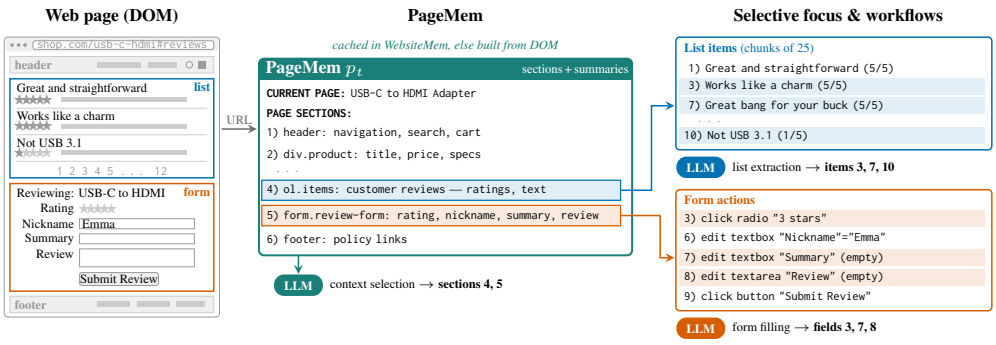


Figure 1: Overview of WebChallenger. (left) The DOM of a page is split into sections that correspond to its semantic regions, here a review list and a review form. (middle) The sections, each with a one-sentence summary, form the PageMem, which is cached in the website memory; the agent skims the summaries and selects only the task-relevant sections. (right) Each selected section is expanded and handled by a workflow chosen by its type, so filtering a long list or filling a whole form costs the agent one action.

**Hierarchy.** PageMem has four levels: a *WebsiteMem* $\mathcal{M}_w$ holds all PageMems of a website $w$; a *PageMem* $p$ holds one page's title, URL, summary, and ordered sections $(s_1, \ldots, s_n)$; a *PageSection* $s_i$ maps to a DOM sub-tree (a navigation bar, a product listing, a review form) and carries DOM-derived attributes (tag, class, bounding box) plus mutable metadata (summary, extracted details); and an *Element* $e$ is one interactive widget, with the attributes needed to build a selector and state such as its value and dropdown items.

**Construction.** PageSections are produced by recursively splitting the DOM tree, stopping at nodes that fall below a size threshold, so that a section's full content fits in one LLM call, or that match a grouping tag (`form`, `ul`, `li`, `table`, `section`, etc.); consecutive siblings sharing tag and class are merged into one unbounded *list section*, represented as a sequence of uniform sub-sections, one per item. Clickable elements are identified with heuristics adapted from the BrowserUse library (Müller & Žunic., 2024) and assigned to their ancestor section, and an LLM or VLM writes a one-sentence summary for each section and for the page (algorithm in Appendix A.1).

### 3.3 Exploration and Memory

Before any task is attempted, an offline exploration phase traverses each target website $w$ and produces the WebsiteMem $\mathcal{M}_w$ used at inference. Exploration is deterministic and needs no LLM guidance, demonstrations, or documentation, so a site's structure is available from the first task at a fixed one-time cost (details in Appendix A.2).

**Traversal.** Starting from the homepage, exploration clicks every unique clickable element in document order, exploring a single item of any repeated structure such as a list of results. Each click is recorded as a state transition: a navigation to an unexplored page adds that URL to the frontier, and a change to the current page, such as a menu expanding, stores the revealed elements as the clicked element's dropdown items. Elements already explored elsewhere on the site are skipped, and the procedure repeats depth-first on the frontier, summarizing each visited page, until it reaches a depth, element, page, or time budget (values in Appendix B).

**Use at inference.** $\mathcal{M}_w$ is saved per site as JSON and reused across tasks without ever being loaded into the model's context: at task start the agent may select a small bookmark set $B_\tau \subseteq \mathcal{M}_w$ that stays available as navigation shortcuts, and the section summaries and dropdown contents recorded during exploration are reused by the observation pipeline (§3.4) without re-opening any menu.

### 3.4 DIVIDE-AND-CONQUER OBSERVATION

A full accessibility tree can exceed the reliable context window of a small LLM, and even when it fits, the task-relevant signal is diluted among boilerplate; the observation pipeline therefore reads a page through several focused sub-prompts and condenses the result into a summarized observation $\hat{o}_t$.

**PageMem retrieval and update.** For the current page $p_t$ the agent reuses the PageMem cached in $\mathcal{M}_w$ from exploration or from an earlier visit in the same task, and builds a fresh one from the live DOM otherwise; in a reused PageMem only sections whose elements have changed are re-summarized, which amortizes summarization across timesteps and repeat visits.

**Section selection.** The LLM is shown the list of section summaries for $p_t$ along with the task instruction $I$ and the interaction history $h_t$, and returns a subset of sections $S_t \subseteq \{s_1, \ldots, s_n\}$ judged relevant to the task.

**Detail extraction.** For each selected section $s \in S_t$ the LLM extracts task-relevant information from the section's full content (accessibility sub-tree and page metadata), with VLM descriptions of any visible image above a minimum size included in the prompt. For a table or list section, extraction is preceded by item selection: items are grouped into chunks of at most 25, the LLM selects the relevant items from each chunk, and only those reach the extraction prompt (Figure 1, middle-right), which keeps even very long lists within context. Extractions are cached on the section, keyed by the exact content string they were computed from, so a cache hit implies identical input (Appendix E.3).

**Summary synthesis.** Finally the LLM condenses the extractions from all selected sections into a one-paragraph summary $\hat{o}_t$, which is passed to the action module and appended to the history $h_t$; one paragraph captures the task-relevant content of a page in most cases while keeping the history compact (full process in Appendix A.3).

### 3.5 COMPOUND ACTIONS AND WORKFLOWS

At each timestep the action module selects $a_t$ from a candidate set $\mathcal{A}_t$ assembled from the current PageMem, the selected sections $S_t$, and the agent's memory. Rather than an LLM tool-use interface, $\mathcal{A}_t$ is presented as a numbered list and the model returns the index of its choice, which we find more reliable for small open-weight models (Appendix E.4). Many actions are *compound*: executing one runs a workflow that combines LLM sub-calls with browser operations (implemented with Playwright) to complete a multi-step interaction as a single action.

**Action selection.** $\mathcal{A}_t$ has three groups: *navigation* actions (previously visited URLs, bookmarks in $B_\tau$, a type-URL action, and where applicable switch-tab and switch-website), *element* actions gathered from $S_t$ after a rule-based pre-filter removes no-ops such as navigating to the current page or clicking a selected radio button, and *end task*, which is always available. The LLM first filters the list to the candidates most promising for the task and then selects $a_t$ among them.

**Workflows.** Workflows follow one rule: an interaction whose intermediate steps change only *part* of the current page (a dropdown expanding, search suggestions appearing, fields being filled one by one) is collapsed into a single compound action, while an action that navigates to a different page remains an individual agent decision. Two representative workflows follow; Appendix A.5 lists the full set.

*Dropdown selection.* The workflow clicks the dropdown element, extracts the list of revealed options via a section-level diff, prompts the LLM to choose one option by index, and clicks the chosen option.

*Form submission.* The LLM selects which fields to fill, is prompted for each field's value, which is entered on the page, and finally reviews the completed form to edit further or submit (Figure 1, bottom-right), so the agent-level action is a single `SubmitForm` step.

**End-task.** The end-task action invokes a one-time verification call in which the LLM, given the task and history, either produces the final answer or reports the task incomplete, in which case it is re-prompted to select a different action.

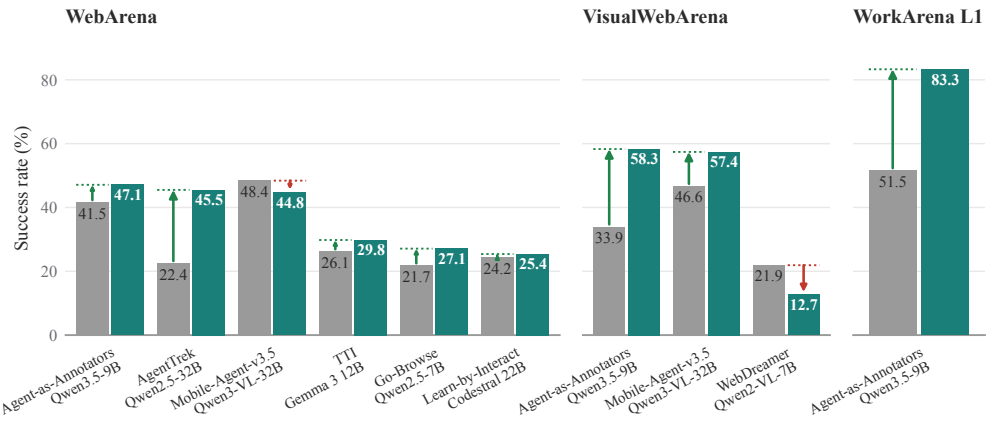


Figure 2: Harness versus fine-tuning. Gray: the published success of a fine-tuned small open-weight agent. Teal: the same base model, without any fine-tuning, run inside the WebChallenger harness. Arrows mark the change. The full comparison, including agents built on proprietary and >120B models, is in Table 5 (Appendix B).

## 4 Experiments

**Benchmarks and setup.** **WebArena** (Zhou et al., 2024) has 812 tasks in 6 simulated environments that mimic common website types (forum, wiki, shopping) with programmatic and LLM evaluation; **VisualWebArena** (Koh et al., 2024) builds on it with 910 tasks that require visual reasoning; **Online-Mind2Web** (Xue et al., 2025) has 300 tasks on 136 live websites, which we score by human evaluation; and **WorkArena** (Drouin et al., 2024) has 330 enterprise tasks on complex user interfaces. The four-benchmark evaluation runs locally on a single consumer NVIDIA RTX 3090 GPU with off-the-shelf models and no fine-tuning: `GLM-4-32B-0414` (Z.ai, 2025) as the LLM controller and `Qwen2.5-VL-7B-Instruct` (Bai et al., 2025b) (`Qwen3-VL-4B-Instruct` (Bai et al., 2025a) for VisualWebArena) for image captioning, with the same prompts throughout and temperature 0. Each benchmark's websites are explored once before inference, and the agent's memory is reset to the post-exploration state between tasks (Appendix B).

### 4.1 Harness versus Fine-Tuning

Figure 2 asks the question a small-model practitioner faces: given a base model, is it better to fine-tune it for web navigation or to put it in a better harness? For every fine-tuned small open-weight agent with a reported WebArena or VisualWebArena result (Table 1), we run its base model as released inside WebChallenger with no fine-tuning and no change to prompts or hyperparameters. The harness recovers what fine-tuning bought in most cases and often exceeds it: Qwen3.5-9B rises from 41.5 to 47.1 on WebArena, from 33.9 to 58.3 on VisualWebArena, and from 51.5 to 83.3 on WorkArena L1, Qwen2.5-32B from 22.4 to 45.5, and the three smallest text models (Codestral 22B, Qwen2.5-7B, Gemma 3 12B) gain 1–5 points. The exceptions are Qwen3-VL-32B on WebArena, whose fine-tuned agent, Mobile-Agent-v3.5, is trained end-to-end on GUI trajectories that our text-first pipeline does not exploit and which loses 3.6 points, and the 7B Qwen2-VL on VisualWebArena, which is too weak to follow the pipeline's structured prompts (§5). With our own configuration (GLM-4-32B with a 7B vision model) WebChallenger reaches 56.3% on WebArena, 48.7% on VisualWebArena, 51.0% on Online-Mind2Web, and 70.9% on WorkArena, the highest of any small open-weight agent on all four (Table 5). Agents built on proprietary or >120B models appear in Table 5 (appendix), since their cost

Table 1: Fine-tuned small open-weight agents in Figure 2 and the data each is fine-tuned on, as described in the cited paper; base models are named in the figure.

| Agent | Fine-tuned on |
|---|---|
| Agent-as-Annotators (Lù & Reddy, 2026) | distilled from Gemini 3 Pro |
| AgentTrek (Xu et al., 2025) | web-tutorial replays |
| Mobile-Agent-v3.5 (Xu et al., 2026) | GUI trajectories + RL |
| TTI (Shen et al., 2025) | curriculum online RL |
| Go-Browse (Gandhi & Neubig, 2025) | structured exploration |
| Learn-by-Interact (Su et al., 2025) | synthesized from docs |
| WebDreamer (Gu et al., 2025) | world-model synthesis |

Table 2: Harness and backbone comparisons on WebArena-lite (success rate, %). Upper block: backbone fixed, harness varied, against BrowserGym's GenericAgent (Chezelles et al., 2025) and WebTactix (WebTactix). Lower block: harness fixed, backbone varied. Shading marks the best and second-best value per column within each block.

| Harness | Backbone | Shopping | Reddit | GitLab | Maps | CMS | Avg |
|---|---|---|---|---|---|---|---|
| *Harness varied, backbone fixed* | | | | | | | |
| GenericAgent | GLM-4-32B | 19.6 | 23.8 | 28.1 | 16.1 | 11.4 | 19.4 |
| *WebChallenger* | GLM-4-32B | 56.5 | 71.4 | 65.6 | 45.2 | 60.0 | 58.8 |
| WebTactix | GLM-5.3-Flash | 50.0 | 76.2 | 50.0 | 32.3 | 62.9 | 52.7 |
| *WebChallenger* | GLM-5.3-Flash | 58.7 | 85.7 | 78.1 | 48.4 | 71.4 | 66.7 |
| WebTactix | Gemini 3.8 Flash | 50.0 | 81.0 | 65.6 | 61.3 | 65.7 | 62.4 |
| *WebChallenger* | Gemini 3.8 Flash | 73.9 | 90.5 | 93.8 | 61.3 | 65.7 | 75.8 |
| *Backbone varied, harness fixed* | | | | | | | |
| *WebChallenger* | GPT-4o-mini | 37.0 | 57.1 | 43.8 | 51.6 | 51.4 | 46.7 |
| *WebChallenger* | GLM-4-32B | 56.5 | 71.4 | 65.6 | 45.2 | 60.0 | 58.8 |
| *WebChallenger* | GPT-5 | 60.9 | 71.4 | 68.8 | 74.2 | 71.4 | 68.5 |

Table 3: Component ablations on WebArena-lite (165 tasks, GLM-4-32B): success, change $\Delta$ from the full system, and cost in agent steps and GLM-4-32B tokens (Tok./prompt: mean input per LLM call; a compound action is one step). Per-site results: Table 9. Shading: best and second-best per column.

| Method | Success (%) | $\Delta$ | Steps | Tokens | Tok./prompt |
|---|---|---|---|---|---|
| *WebChallenger* | 58.8 | – | 7.2 | 47.0M | 1850 |
| − memory | 50.9 | −7.9 | 7.3 | 48.1M | 1781 |
| − compound actions | 49.1 | −9.7 | 9.9 | 64.9M | 2109 |
| − observation pipeline | 41.8 | −17.0 | 11.3 | 36.0M | 8793 |

per task is one to two orders of magnitude higher and a small model in our harness already matches them: Qwen3.5-9B (47.1%) outperforms every GPT-4o-based agent on WebArena but ScribeAgent (53.0%), which pairs GPT-4o with a fine-tuned 32B model, and Qwen3-VL-32B (57.4%) outperforms WALT with GPT-5 (52.9%) on VisualWebArena. Online-Mind2Web, whose 136 live websites were never used during development, is the strongest test of generality: the same configuration reaches 51.0%, ahead of every fine-tuned small agent, so WebChallenger exploits structure shared across websites rather than properties of particular sites. The harness, not fine-tuning or model scale, is thus the main lever on long-horizon web tasks.

### 4.2 Harness versus Harness at a Fixed Backbone

Table 2 holds the backbone fixed and varies only the harness, which isolates our contribution from the model, on the 165-task WebArena-lite subset (Liu et al., 2024), where our score (58.8) tracks the full WebArena score (56.3). GenericAgent, the reference harness of BrowserGym, the platform these benchmarks run on, is run with our own GLM-4-32B; WebTactix, the highest-scoring agent on the official WebArena leaderboard at the time of evaluation (WebArena Team, 2026), is run with two inexpensive API models, GLM-5.3-Flash and Gemini 3.8 Flash, so that both harnesses could be evaluated on the full subset at equal settings (Appendix B.2). With GLM-4-32B, WebChallenger triples the success of GenericAgent (58.8% vs. 19.4%), and with GPT-oss-120b held fixed on WorkArena L1 it reaches 77.4% against GenericAgent's 50.9% (Table 5); against WebTactix it gains 14 points with GLM-5.3-Flash and 13 with Gemini 3.8 Flash, improving on every site but Maps and CMS with the latter. The lower block varies the backbone inside WebChallenger: a cheap API model (GPT-4o-mini) loses 12 points relative to GLM-4-32B and a frontier model (GPT-5) gains 10, so the harness benefits from stronger models while keeping most of its capability with weaker ones.

**Which component matters, and what does it cost?** We remove each component in turn (Table 3): *memory* (bookmarks, dropdown contents, and pre-cached summaries; PageMem is still built at inference, from scratch for every task), *compound actions* (the agent is restricted to single basic actions plus navigation, so search, dropdown, and form-filling workflows disappear), and the *observation pipeline* (section selection and detail extraction are replaced by one prompt holding the full

accessibility tree and all actions, with history reduced to a list of prior actions, the standard harness design). Every component helps, and the observation pipeline helps most: removing it costs 17 points on average, and on the two environments with the largest and most cluttered pages it cuts success from 45.2 to 25.8 on Maps and from 60.0 to 25.7 on CMS. Compound actions matter most on CMS ($-20$ points), whose tasks revolve around forms and menus; memory matters most on Shopping and CMS, the sites with the deepest navigation, and is within noise on Reddit and Maps (Appendix B.4). Single-factor ablations (Appendix E.4) separate the confounded pieces: removing the observation pipeline while keeping the summarized history costs the same 17.0 points, so the drop is entirely due to reading whole pages rather than to the history format; within memory, the recorded dropdown contents account for the whole effect ($-7.9$, matching the full memory ablation) and bookmarks for less ($-3.6$); dropping detail extraction costs 7.3 points; and replacing the numbered-list action interface with native tool calling costs 6.7. With 165 tasks one standard error of a success rate is about four points, so the bookmark effect is within one standard error, while every other difference in this section exceeds it. The observation pipeline trades inference compute for accuracy: without it the agent uses fewer tokens in total (36.0M vs. 47.0M) but each prompt becomes $4.8\times$ longer and the agent needs half again as many steps, because it now reasons over whole pages and makes more mistakes. Compound actions make the agent cheaper as well as better: removing them raises total tokens by $38\%$ and steps from 7.2 to 9.9, since each interaction a workflow completed internally now needs its own observe-and-decide cycle. Offline exploration is a one-time cost of a few tens of millions of summarization tokens per benchmark, reused across every task (Appendix B.3).

### 4.3 WebChallenger+: The Same Decomposition on Frontier VLMs

WebChallenger separates text from vision because a small text model cannot read a rendered page: a separate vision model is invoked afresh on every screenshot and the text model is handed summaries it can act on. A frontier VLM reads the page natively and holds a task's history in context, so everything the pipeline delegated to a separate model can be done by the one model; **WebChallenger+** is this instantiation. It keeps the two principles the paper is about, structural page decomposition and selective observation, and shifts each mechanism one step toward the stronger model: sections are presented as an outline of the page's structure rather than as model-written summaries, which existed to spare a small model the raw structure; the offline memory is dropped, since a million-token context holds what a small model had to be reminded of; and each decision selects one browser tool that may perform several elementary actions, since the model can manage its own context; two of the three mechanisms thus remain in lighter form and one is removed (Appendix F). WebChallenger+ is therefore evidence that the decomposition transfers rather than an ablation; the bridge configuration that isolates the model change, the original pipeline with Gemini 3.8 Flash on the same tasks, appears as the ▲ points in Figure 3.

**Setting.** The comparators are the harnesses these models ship with and are reported with: Claude Code (Fable 5.1) and Codex (GPT-5.6 Sol, GPT-6 Astra, and Gemini 3.8 Flash), both of which drive a real Chrome instance through their tool interface (Appendix E.5 discusses the choice). We evaluate on 40 WorkArena L3 procedures, multi-page enterprise procedures such as onboarding a user or reconciling expenses that are far harder than the L1 tasks in Table 5, and on 45 WebArena tasks. Running seven frontier-agent configurations on the full benchmarks was beyond our API budget, so we sampled subsets at random, stratified by task family (WorkArena) and by website and difficulty (WebArena), kept the infeasible tasks because recognizing infeasibility is part of the task, and release the subsets with our run configuration.[1] All systems receive the same task text and instructions, run one session per task, and are scored by the benchmarks' own evaluators; we report success and mean estimated inference cost per task, including failed tasks (Appendix F).

**Results.** On every backbone we tried, WebChallenger+ reaches the success of the model's own harness at a fraction of its cost (Figure 3). On subsets of 40 and 45 tasks one task is 2.2–2.5 points and one standard error about four to six points, so success differences of one or two tasks are within noise; the cost differences, of $1.4$–$4\times$, are not. On WorkArena L3 it solves all 40 procedures with GPT-6 Astra at \$1.48 per task, where Codex needs \$4.06 for the same result, and 39 of 40 with Fable 5.1 at \$1.21, where Claude Code solves 36 at \$1.64; with Gemini 3.8 Flash it solves 38 at \$0.21. On WebArena it reaches 36 of 45 with Fable 5.1 or GPT-6 Astra at \$0.42 and \$0.65, level with

[1] Link withheld for double-blind review; the subsets and configuration are released with the code.

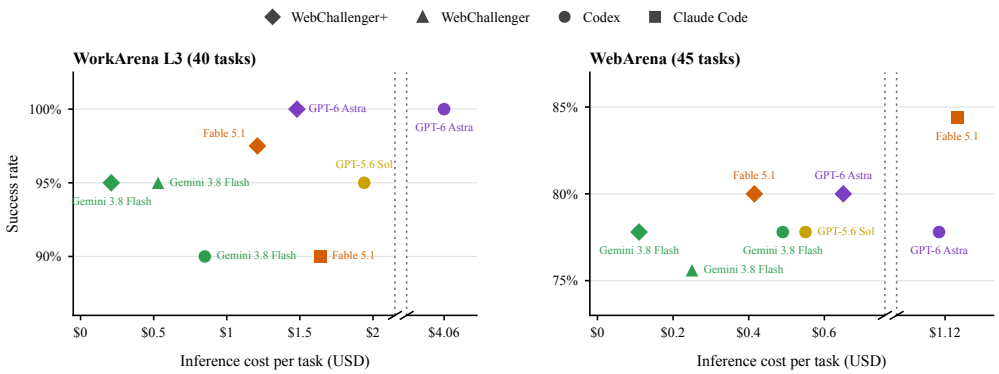


Figure 3: Success against estimated inference cost per task on WorkArena L3 (left) and WebArena (right). Marker shape encodes the harness, each point is labelled with its backbone, and the x axis is broken to bring the two most expensive runs into view. On the same backbone, WebChallenger+ (⧫) matches the success of Claude Code and Codex at a fraction of the cost (numbers in Table 13).

every Codex configuration (35) within noise; Claude Code reaches 38 of 45 on this subset at 2.7× the cost of WebChallenger+ on the same backbone. Holding the backbone fixed, WebChallenger+ matches Codex on both benchmarks with Gemini 3.8 Flash and GPT-6 Astra at 1.7–4.4× lower cost and Claude Code with Fable 5.1 at 1.4–2.7× lower cost, repeating the matched-backbone finding of Table 2 with frontier models. The bridge points isolate the pipeline change: the original WebChallenger pipeline with Gemini 3.8 Flash reaches the same 38 of 40 on WorkArena L3 and 34 of 45 on WebArena, but at 2.3–2.5× the cost of WebChallenger+, so the leaner single-model pipeline keeps the accuracy of the decomposition while removing most of its overhead. On a fixed model cost is proportional to the tokens processed, and Appendix F shows where the gap comes from: WebChallenger+ keeps the model context two to four times smaller than the native harnesses.

## 5 Limitations

WebChallenger relies on hand-designed components that encode priors about how web pages are organized, from DOM-based sectioning to a fixed set of workflows, so performance may degrade on sites that diverge from common patterns. It reads a page through many short model calls rather than one long one, which multiplies sequential round trips per task, pays for accuracy in tokens (Table 3), and needs a model that follows structured output formats reliably: the 7B Qwen2-VL falls below its fine-tuned counterpart on VisualWebArena for this reason (Figure 2). Our component ablations each remove more than one thing at once, so their numbers are upper bounds on each component's own contribution (single-factor ablations in Appendix E.4), and all runs are single-seed at temperature 0 (Appendix E.6). The WebChallenger+ comparison uses fixed task subsets and provider list prices, so its cost ratios are indicative rather than precise, and it compares against the harnesses the models ship with rather than every deployed browser agent (Appendix E.5). Finally, all of our evaluation is conducted on benign tasks, and the system's robustness to adversarial page content is uncharacterized (Tur et al., 2025; Zheng et al., 2025b; Xiang et al., 2025; Wu et al., 2026; Ying et al., 2026; Zhang et al., 2025a;c; Wu et al., 2025; Liao et al., 2026; Kuntz et al., 2025; Anthropic, 2025).

## 6 Conclusion

WebChallenger coordinates selective observation, persistent website memory, and compound actions through PageMem, a shared structural representation of web pages; without fine-tuning it sets the state of the art among small open-weight agents on four benchmarks, and matched-backbone comparisons show that the gains come from how the agent observes, remembers, and acts. The same decomposition carries over to frontier VLMs, where WebChallenger+ matches the harnesses those models ship with at a fraction of their cost, so structured, selective observation remains a source of efficiency as models grow.

## Reproducibility Statement

The algorithms behind every component are given in Appendix A (PageMem construction, exploration, the observation pipeline, the agent loop, and the workflows) and every prompt in Appendix D. Appendix B lists all hyperparameters, the benchmark adaptations shared by every system we ran, the settings of the harness comparison, and compute and exploration coverage; Appendix E.6 gives model serving, decoding, evaluation, and state-reset details; and Appendix F lists the exact WebChallenger+ task subsets, execution protocol, model configurations, and cost accounting. Code, exploration memories, task manifests, per-task outcomes, and token logs will be released with the camera-ready version.

## The Use of Large Language Models

Large language models and vision-language models are components of the systems evaluated in this paper, as described in §4 and Appendix F. In preparing the manuscript, LLMs were used to polish wording and to check that numbers quoted in the text agree with the tables and figures; the research questions, system design, experiments, analyses, and claims are the authors' own, and the authors reviewed and take responsibility for all text.

## References

Tamer Abuelsaad, Deepak Akkil, Prasenjit Dey, Ashish Jagmohan, Aditya Vempaty, and Ravi Kokku. Agent-e: From autonomous web navigation to foundational design principles in agentic systems, 2024. URL https://arxiv.org/abs/2407.13032.

Anthropic. Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku, 2024. URL https://www.anthropic.com/news/3-5-models-and-computer-use. Accessed 2026-09-26.

Anthropic. Mitigating the risk of prompt injections in browser use. https://www.anthropic.com/news/prompt-injection-defenses, November 2025. Accessed: 2026-05-11.

Ahmed Awadallah, Yash Lara, Raghav Magazine, Hussein Mozannar, Akshay Nambi, Yash Pandya, Aravind Rajeswaran, Corby Rosset, Alexey Taymanov, Vibhav Vineet, Spencer Whitehead, and Andrew Zhao. Fara-7b: An efficient agentic model for computer use, 2025. URL https://arxiv.org/abs/2511.19663.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, Mei Li, Kaixin Li, Zicheng Lin, Junyang Lin, Xuejing Liu, Jiawei Liu, Chenglong Liu, Yang Liu, Dayiheng Liu, Shixuan Liu, Dunjie Lu, Ruilin Luo, Chenxu Lv, Rui Men, Lingchen Meng, Xuancheng Ren, Xingzhang Ren, Sibo Song, Yuchong Sun, Jun Tang, Jianhong Tu, Jianqiang Wan, Peng Wang, Pengfei Wang, Qiuyue Wang, Yuxuan Wang, Tianbao Xie, Yiheng Xu, Haiyang Xu, Jin Xu, Zhibo Yang, Mingkun Yang, Jianxin Yang, An Yang, Bowen Yu, Fei Zhang, Hang Zhang, Xi Zhang, Bo Zheng, Humen Zhong, Jingren Zhou, Fan Zhou, Jing Zhou, Yuanzhi Zhu, and Ke Zhu. Qwen3-vl technical report, 2025a. URL https://arxiv.org/abs/2511.21631.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-vl technical report, 2025b. URL https://arxiv.org/abs/2502.13923.

Browserbase. Stagehand: The AI browser automation framework, 2024. URL https://github.com/browserbase/stagehand. Accessed 2026-09-26.

Howard Chen, Ramakanth Pasunuru, Jason Weston, and Asli Celikyilmaz. Walking down the memory maze: Beyond context limit through interactive reading, 2023. URL https://arxiv.org/abs/2310.05029.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. Automanual: Constructing instruction manuals by llm agents via interactive environmental learning, 2024. URL https://arxiv.org/abs/2405.16247.

Jiali Cheng, Anjishnu Kumar, Roshan Lal, Rishi Rajasekaran, Hani Ramezani, Omar Zia Khan, Oleg Rokhlenko, Sunny Chiu-Webster, Gang Hua, and Hadi Amiri. Webatlas: An llm agent with experience-driven memory and action simulation, 2025. URL https://arxiv.org/abs/2510.22732.

Thibault Le Sellier De Chezelles, Maxime Gasse, Alexandre Drouin, Massimo Caccia, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Quentin Cappart, Graham Neubig, Ruslan Salakhutdinov, Nicolas Chapados, and Alexandre Lacoste. The browsergym ecosystem for web agent research, 2025. URL https://arxiv.org/abs/2412.05467.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web, 2023. URL https://arxiv.org/abs/2306.06070.

Robert B. Doorenbos, Oren Etzioni, and Daniel S. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the First International Conference on Autonomous Agents*, AGENTS '97, pp. 39–48, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897918770. doi: 10.1145/267658.267666. URL https://doi.org/10.1145/267658.267666.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. Workarena: How capable are web agents at solving common knowledge work tasks?, 2024. URL https://arxiv.org/abs/2403.07718.

Quentin Feuillade--Montixi. WebFurl: A browser-use AI agent with compressed unfoldable HTML representation for high token efficiency. https://github.com/WeaveMindAI/Webfurl, 2026. GitHub repository.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. Autoguide: Automated generation and selection of context-aware guidelines for large language model agents, 2024. URL https://arxiv.org/abs/2403.08978.

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, 2024. URL https://arxiv.org/abs/2305.11854.

Apurva Gandhi and Graham Neubig. Go-browse: Training web agents with structured exploration, 2025. URL https://arxiv.org/abs/2506.03533.

Google DeepMind. Introducing the Gemini 2.5 Computer Use model, 2025. URL https://blog.google/technology/google-deepmind/gemini-computer-use-model/. Accessed 2026-09-26.

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, 2025. URL https://arxiv.org/abs/2410.05243.

Yu Gu, Kai Zhang, Yuting Ning, Boyuan Zheng, Boyu Gou, Tianci Xue, 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, 2025. URL https://arxiv.org/abs/2411.06559.

Izzeddin Gur, Ulrich Rueckert, Aleksandra Faust, and Dilek Hakkani-Tur. Learning to navigate the web, 2018. URL https://arxiv.org/abs/1812.09195.

Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. A real-world webagent with planning, long context understanding, and program synthesis, 2024. URL https://arxiv.org/abs/2307.12856.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models, 2024. URL https://arxiv.org/abs/2401.13919.

Kaiwen He, Zhiwei Wang, Chenyi Zhuang, and Jinjie Gu. Recon-act: A self-evolving multi-agent browser-use system via web reconnaissance, tool generation, and task execution, 2025. URL https://arxiv.org/abs/2509.21072.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxuan Zhang, Juanzi Li, Bin Xu, Yuxiao Dong, Ming Ding, and Jie Tang. Cogagent: A visual language model for gui agents, 2024. URL https://arxiv.org/abs/2312.08914.

Lawrence Keunho Jang, Jing Yu Koh, Daniel Fried, and Ruslan Salakhutdinov. Odysseys: Benchmarking web agents on realistic long horizon tasks, 2026. URL https://arxiv.org/abs/2604.24964.

Dulhan Jayalath, James Bradley Wendt, Nicholas Monath, Sandeep Tata, and Beliz Gunel. Prism: Efficient long-range reasoning with short-context llms, 2025. URL https://arxiv.org/abs/2412.18914.

Imene Kerboua, Sahar Omidi Shayegan, Megh Thakkar, Xing Han Lù, Léo Boisvert, Massimo Caccia, Jérémy Espinas, Alexandre Aussem, Véronique Eglin, and Alexandre Lacoste. Focusagent: Simple yet effective ways of trimming the large context of web agents, 2025. URL https://arxiv.org/abs/2510.03204.

Jihyung Kil, Chan Hee Song, Boyuan Zheng, Xiang Deng, Yu Su, and Wei-Lun Chao. Dual-view visual contextualization for web navigation, 2024. URL https://arxiv.org/abs/2402.04476.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks, 2023. URL https://arxiv.org/abs/2303.17491.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks, 2024. URL https://arxiv.org/abs/2401.13649.

Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents, 2025. URL https://arxiv.org/abs/2407.01476.

Thomas Kuntz, Agatha Duzan, Hao Zhao, Francesco Croce, Zico Kolter, Nicolas Flammarion, and Maksym Andriushchenko. Os-harm: A benchmark for measuring safety of computer use agents, 2025. URL https://arxiv.org/abs/2506.14866.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Hanyu Lai, Xiao Liu, Iat Long Iong, Shuntian Yao, Yuxuan Chen, Pengbo Shen, Hao Yu, Hanchen Zhang, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. Autowebglm: A large language model-based web navigating agent, 2024. URL https://arxiv.org/abs/2404.03648.

Dongjun Lee, Juyong Lee, Kyuyoung Kim, Jihoon Tack, Jinwoo Shin, Yee Whye Teh, and Kimin Lee. Learning to contextualize web pages for enhanced decision making by llm agents, 2025. URL https://arxiv.org/abs/2503.10689.

Kuang-Huei Lee, Xinyun Chen, Hiroki Furuta, John Canny, and Ian Fischer. A human-inspired reading agent with gist memory of very long contexts, 2024. URL https://arxiv.org/abs/2402.09727.

Tao Li, Gang Li, Zhiwei Deng, Bryan Wang, and Yang Li. A zero-shot language agent for computer control with structured reflection, 2023. URL https://arxiv.org/abs/2310.08740.

Zeyi Liao, Jaylen Jones, Linxi Jiang, Yuting Ning, Eric Fosler-Lussier, Yu Su, Zhiqiang Lin, and Huan Sun. Redteamcua: Realistic adversarial testing of computer-use agents in hybrid web-os environments, 2026. URL https://arxiv.org/abs/2505.21936.

Genglin Liu, Shijie Geng, Sha Li, Hejie Cui, Sarah Zhang, Xin Liu, and Tianyi Liu. Webcoach: Self-evolving web agents with cross-session memory guidance, 2025. URL https://arxiv.org/abs/2511.12997.

Xiao Liu, Tianjie Zhang, Yu Gu, Iat Long Iong, Yifan Xu, Xixuan Song, Shudan Zhang, Hanyu Lai, Xinyi Liu, Hanlin Zhao, Jiadai Sun, Xinyue Yang, Yu Yang, Zehan Qi, Shuntian Yao, Xueqiao Sun, Siyi Cheng, Qinkai Zheng, Hao Yu, Hanchen Zhang, Wenyi Hong, Ming Ding, Lihang Pan, Xiaotao Gu, Aohan Zeng, Zhengxiao Du, Chan Hee Song, Yu Su, Yuxiao Dong, and Jie Tang. Visualagentbench: Towards large multimodal models as visual foundation agents, 2024. URL https://arxiv.org/abs/2408.06327.

Xing Han Lù and Siva Reddy. Structured distillation of web agent capabilities enables generalization, 2026. URL https://arxiv.org/abs/2604.07776.

Tiange Luo, Lajanugen Logeswaran, Justin Johnson, and Honglak Lee. Visual test-time scaling for gui agent grounding, 2025. URL https://arxiv.org/abs/2505.00684.

Xing Han Lù, Zdeněk Kasner, and Siva Reddy. Weblinx: Real-world website navigation with multi-turn dialogue, 2024. URL https://arxiv.org/abs/2402.05930.

Kenneth Marino and Ana Marasović. Computer use survey: A visual survey of computer use agents, 2025. URL https://kennethmarino.com/computeruse/computeruse.html.

Sami Marreed, Alon Oved, Avi Yaeli, Segev Shlomov, Ido Levy, Offer Akrabi, Aviad Sela, Asaf Adi, and Nir Mashkif. Towards enterprise-ready computer using generalist agent, 2025. URL https://arxiv.org/abs/2503.01861.

Atsuyuki Miyai, Zaiying Zhao, Kazuki Egashira, Atsuki Sato, Tatsumi Sunada, Shota Onohara, Hiromasa Yamanishi, Mashiro Toyooka, Kunato Nishina, Ryoma Maeda, Kiyoharu Aizawa, and Toshihiko Yamasaki. Webchorearena: Evaluating web browsing agents on realistic tedious web tasks, 2025. URL https://arxiv.org/abs/2506.01952.

Hans Moravec. *Mind Children: The Future of Robot and Human Intelligence*. Harvard University Press, Cambridge, MA, 1988.

Magnus Müller and Gregor Žunic. Browser use: Enable ai to control your browser, 2024. URL https://github.com/browser-use/browser-use.

Hadi Nekoei, Aman Jaiswal, Patrice Bechard, Oleh Shliazhko, Orlando Marquez Ayala, Mathieu Reymond, Massimo Caccia, Alexandre Drouin, Sarath Chandar, and Alexandre Lacoste. Jefhinter: Leveraging offline knowledge for improving web agents adaptation, 2025. URL https://arxiv.org/abs/2510.04373.

OpenAI. Introducing operator, 2025. URL https://openai.com/index/introducing-operator/.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory, 2025. URL https://arxiv.org/abs/2509.25140.

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://arxiv.org/abs/2502.11357.

Xinyu Pang, Ruixin Hong, Hongming Zhang, and Changshui Zhang. Assimilation and accommodation: Task-adaptive hierarchical abstraction for solving web tasks. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 14000–14014, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.720. URL https://aclanthology.org/2025.findings-acl.720/.

Joonhyung Park, Peng Tang, Sagnik Das, Srikar Appalaraju, Kunwar Yashraj Singh, R. Manmatha, and Shabnam Ghadar. R-vlm: Region-aware vision language model for precise gui grounding, 2025. URL https://arxiv.org/abs/2507.05673.

Viraj Prabhu, Yutong Dai, Matthew Fernandez, Jing Gu, Krithika Ramakrishnan, Yanqi Luo, Silvio Savarese, Caiming Xiong, Junnan Li, Zeyuan Chen, and Ran Xu. Walt: Web agents that learn tools, 2025. URL https://arxiv.org/abs/2510.01524.

Aini Putkonen, Aurélien Nioche, Markku Laine, Crista Kuuramo, and Antti Oulasvirta. Fragmented visual attention in web browsing: Weibull analysis of item visit times. In Jaap Kamps, Lorraine Goeuriot, Fabio Crestani, Maria Maistro, Hideo Joho, Brian Davis, Cathal Gurrin, Udo Kruschwitz, and Annalina Caputo (eds.), *Advances in Information Retrieval*, pp. 62–78, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-28238-6.

Gabriel Sarch, Lawrence Jang, Michael J. Tarr, William W. Cohen, Kenneth Marino, and Katerina Fragkiadaki. Vlm agents generate their own memories: Distilling experience into embodied programs of thought, 2025a. URL https://arxiv.org/abs/2406.14596.

Gabriel Sarch, Snigdha Saha, Naitik Khandelwal, Ayush Jain, Michael J. Tarr, Aviral Kumar, and Katerina Fragkiadaki. Grounded reinforcement learning for visual reasoning, 2025b. URL https://arxiv.org/abs/2505.23678.

ServiceNow. Browsergym leaderboard, 2025. URL https://huggingface.co/spaces/ServiceNow/browsergym-leaderboard.

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, 2023. URL https://arxiv.org/abs/2306.00245.

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, 2024. URL https://arxiv.org/abs/2411.15004.

Junhong Shen, Hao Bai, Lunjun Zhang, Yifei Zhou, Amrith Setlur, Shengbang Tong, Diego Caples, Nan Jiang, Tong Zhang, Ameet Talwalkar, and Aviral Kumar. Thinking vs. doing: Agents that reason by scaling test-time interaction, 2025. URL https://arxiv.org/abs/2506.07976.

Kunal Singh, Shreyas Singh, and Mukund Khanna. Trishul: Towards region identification and screen hierarchy understanding for large vlm based gui agents, 2025. URL https://arxiv.org/abs/2502.08226.

Skyvern AI. Skyvern: Automate browser-based workflows with AI, 2024. URL https://github.com/Skyvern-AI/skyvern. Accessed 2026-09-26.

Yueqi Song, Frank Xu, Shuyan Zhou, and Graham Neubig. Beyond browsing: Api-based web agents, 2025. URL https://arxiv.org/abs/2410.16464.

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

Yu Su. Computer use: Modern moravec's paradox. Yu's Substack, September 2025. URL https://yusu.substack.com/p/computer-use-modern-moravecs-paradox. Blog post, accessed May 7, 2026.

Ada Defne Tur, Nicholas Meade, Xing Han Lù, Alejandra Zambrano, Arkil Patel, Esin Durmus, Spandana Gella, Karolina Stańczak, and Siva Reddy. Safearena: Evaluating the safety of autonomous web agents, 2025. URL https://arxiv.org/abs/2503.04957.

Gaurav Verma, Rachneet Kaur, Nishan Srishankar, Zhen Zeng, Tucker Balch, and Manuela Veloso. Adaptagent: Adapting multimodal web agents with few-shot learning from human demonstrations, 2024. URL https://arxiv.org/abs/2411.13451.

Zhaoyang Wang, Qianhui Wu, Xuchao Zhang, Chaoyun Zhang, Wenlin Yao, Fazle Elahi Faisal, Baolin Peng, Si Qin, Suman Nath, Qingwei Lin, Chetan Bansal, Dongmei Zhang, Saravan Rajmohan, Jianfeng Gao, and Huaxiu Yao. Webxskill: Skill learning for autonomous web agents, 2026. URL https://arxiv.org/abs/2604.13318.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory, 2024. URL https://arxiv.org/abs/2409.07429.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks, 2025. URL https://arxiv.org/abs/2504.06821.

WebArena Team. WebArena leaderboard, 2026. URL https://docs.google.com/spreadsheets/d/1M801lEpBbKSNwP-vDBkC_pF7LdyGU1f_ufZb_NWNBZQ/edit. Accessed September 19, 2026.

WebTactix. WebTactix: An autonomous LLM web agent. GitHub repository, n.d. URL https://github.com/paper-submission-anoymous/webtactix. Accessed September 19, 2026.

Chen Henry Wu, Rishi Shah, Jing Yu Koh, Ruslan Salakhutdinov, Daniel Fried, and Aditi Raghunathan. Dissecting adversarial robustness of multimodal lm agents, 2025. URL https://arxiv.org/abs/2406.12814.

Xinyi Wu, Geng Hong, Yueyue Chen, MingXuan Liu, Feier Jin, Xudong Pan, Jiarun Dai, and Baojun Liu. When bots take the bait: Exposing and mitigating the emerging social engineering attack in web automation agent, 2026. URL https://arxiv.org/abs/2601.07263.

Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, Dawn Song, and Bo Li. Guardagent: Safeguard llm agents by a guard agent via knowledge-enabled reasoning, 2025. URL https://arxiv.org/abs/2406.09187.

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, Xuhao Hu, Feiyu Gao, Junjie Cao, Zihua Wang, Zhiyuan Chen, Jitong Liao, Qi Zheng, Jiahui Zeng, Ze Xu, Shuai Bai, Junyang Lin, Jingren Zhou, and Ming Yan. Mobile-agent-v3.5: Multi-platform fundamental gui agents, 2026. URL https://arxiv.org/abs/2602.16855.

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. Agenttrek: Agent trajectory synthesis via guiding replay with web tutorials, 2025. URL https://arxiv.org/abs/2412.09605.

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, 2025. URL https://arxiv.org/abs/2504.01382.

Ke Yang, Yao Liu, Sapana Chaudhary, Rasool Fakoor, Pratik Chaudhari, George Karypis, and Huzefa Rangwala. Agentoccam: A simple yet strong baseline for llm-based web agents, 2025. URL https://arxiv.org/abs/2410.13825.

Zonghao Ying, Yangguang Shao, Jianle Gan, Gan Xu, Wenxin Zhang, Quanchen Zou, Junzheng Shi, Zhenfei Yin, Mingchuan Zhang, Aishan Liu, and Xianglong Liu. Securewebarena: A holistic security evaluation benchmark for lvlm-based web agents, 2026. URL https://arxiv.org/abs/2510.10073.

Simon Yu, Gang Li, Weiyan Shi, and Peng Qi. Polyskill: Learning generalizable skills through polymorphic abstraction, 2025. URL https://arxiv.org/abs/2510.15863.

Z.ai. Glm-4-32b-0414, 2025. URL https://huggingface.co/zai-org/GLM-4-32B-0414.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2026. URL https://arxiv.org/abs/2512.24601.

Kaiyuan Zhang, Mark Tenenholtz, Kyle Polley, Jerry Ma, Denis Yarats, and Ninghui Li. Browsesafe: Understanding and preventing prompt injection within ai browser agents, 2025a. URL https://arxiv.org/abs/2511.20597.

Ruichen Zhang, Mufan Qiu, Zhen Tan, Mohan Zhang, Vincent Lu, Jie Peng, Kaidi Xu, Leandro Z. Agudelo, Peter Qian, and Tianlong Chen. Symbiotic cooperation for web agents: Harnessing complementary strengths of large and small llms, 2025b. URL https://arxiv.org/abs/2502.07942.

Yanzhe Zhang, Tao Yu, and Diyi Yang. Attacking vision-language computer agents via pop-ups, 2025c. URL https://arxiv.org/abs/2411.02391.

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, 2024. URL https://arxiv.org/abs/2408.15978.

Minglu Zhao, Dehong Xu, and Tao Gao. From cognition to computation: A comparative review of human attention and transformer architectures, 2024. URL https://arxiv.org/abs/2407.01548.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. Gpt-4v(ision) is a generalist web agent, if grounded, 2024. URL https://arxiv.org/abs/2401.01614.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. Skillweaver: Web agents can self-improve by discovering and honing skills, 2025a. URL https://arxiv.org/abs/2504.07079.

Boyuan Zheng, Zeyi Liao, Scott Salisbury, Zeyuan Liu, Michael Lin, Qinyuan Zheng, Zifan Wang, Xiang Deng, Dawn Song, Huan Sun, and Yu Su. Webguard: Building a generalizable guardrail for web agents, 2025b. URL https://arxiv.org/abs/2507.14293.

Hongbin Zhong, Fazle Faisal, Luis França, Tanakorn Leesatapornwongsa, Adriana Szekeres, Kexin Rong, and Suman Nath. Actionengine: From reactive to programmatic gui agents via state machine memory, 2026. URL https://arxiv.org/abs/2602.20502.

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, 2024. URL https://arxiv.org/abs/2307.13854.

## APPENDIX OVERVIEW

The appendix is organized as follows.

- Appendix A (Implementation details) gives the algorithms and parameters behind the method: PageMem construction (App. A.1, supporting §3.2), offline exploration (App. A.2, §3.3), the observation pipeline (App. A.3, §3.4), the agent loop (App. A.4), and the action workflows with the history format (App. A.5, §3.5).
- Appendix B (Additional experiment details) supports §4 with the full comparison including proprietary and large-model agents (App. B.1), benchmark-specific adaptations and all hyperparameters, the settings of the WebTactix comparison (App. B.2, Table 2), compute and exploration coverage (App. B.3), and per-site ablation results (App. B.4).
- Appendix C (Broader impacts) complements the Limitations in §5.
- Appendix D (Prompts) lists every system prompt used by the observation pipeline, the agent loop, the workflows, and the VLM; the text cites each by name.
- Appendix E (Claims, bounds, comparators, and reproducibility) states what is guaranteed by construction (per-call input bound, cache soundness), reports the single-factor ablations, explains the choice of comparators and configurations (App. E.5), and lists the serving, evaluation, and environment details needed to reproduce every table (App. E.6).
- Appendix F (WebChallenger+ details) gives the task subsets, execution protocol, configurations, and cost accounting for §4.3.

# A Implementation Details

## A.1 PageMem

Each PageMem is built in two stages: DividePage (Algorithm 1) partitions the DOM into empty PageSections, and UpdatePageMem (Algorithm 2) populates them with elements and summaries.

### A.1.1 Memory Structure

Throughout, we use $\sigma$ for model-generated summaries, $\alpha$ for immutable DOM-derived attributes, and $\mu$ for mutable agent-side state accumulated during exploration and task execution. Descriptions of $\alpha$ and $\mu$ below give representative fields rather than exhaustive listings; full field schemas are documented in our released code.

**WebsiteMem.** A WebsiteMem for website $w$ is a tuple

$$\mathcal{M}_w = (P_w,\, T_w,\, E_w),$$

where $P_w$ is a mapping from URL to PageMem, collecting all concrete pages encountered on $w$; $T_w$ is a list of list-page templates, each itself a PageMem, against which newly visited pages are matched by structural comparison (see A.2); and $E_w$ is the set of all elements encountered on $w$, used to deduplicate elements during exploration.

**PageMem.** A PageMem is a tuple

$$p = (u_p,\, n_p,\, \sigma_p,\, S_p,\, \mu_p),$$

where $u_p$ is the page URL; $n_p$ is the page title; $\sigma_p$ is a VLM-generated page-level summary; $S_p = (s_1, \ldots, s_{|S_p|})$ is an ordered list of PageSections; and $\mu_p$ holds page-level agent state (e.g., information extracted by agent, the agent's past interaction history on the page).

**PageSection.** A PageSection is a tuple

$$s = (\sigma_s,\, E_s,\, S'_s,\, \alpha_s,\, \mu_s),$$

where $\sigma_s$ is a VLM-generated section summary; $E_s = (e_1, \ldots, e_{k_s})$ is an ordered list of Elements contained in the section; $S'_s = (s'_1, \ldots, s'_{m_s})$ is an ordered list of sub-sections, empty for normal sections and containing one sub-section per item for list sections; $\alpha_s$ holds DOM-derived attributes used for creating selectors (e.g., id, tag, class, DOM-subtree handle, bounding box) and $\mu_s$ holds mutable agent state (e.g., task-relevant extractions, VLM-generated image descriptions, a staleness flag indicating whether the DOM subtree has changed since $\sigma_s$ was last computed).

**Element.** An Element is a tuple

$$e = (\alpha_e,\, E'_e,\, \mu_e),$$

where $\alpha_e$ holds DOM-derived attributes (e.g., id, tag, class, role, label, type); $E'_e = (e'_1, \ldots, e'_{l_e})$ is an ordered list of dropdown items (themselves Elements), which is empty for non-dropdown elements; and $\mu_e$ holds mutable agent state (e.g., the element's current input value, a flag for whether the agent has clicked the element during the current task).

### A.1.2 Page division.

DividePage takes the root of the DOM tree and returns a PageMem whose ordered section list forms the structural skeleton of the page. The procedure recursively descends the DOM, terminating at nodes that form a meaningful grouping, either semantically (by tag), visually (by size), or structurally (by repetition of siblings). Runs of consecutive siblings sharing tag and class are merged into a list section before recursion; list sections are always terminal and are never further subdivided.

**Parameters.** The grouping tag set is

$$\mathcal{T}_{\text{group}} = \{\texttt{ol}, \texttt{ul}, \texttt{table}, \texttt{form}, \texttt{fieldset}, \texttt{aside}, \texttt{article},\\ \texttt{details}, \texttt{p}, \texttt{img}, \texttt{embed}, \texttt{code}, \texttt{group}, \texttt{nav}, \texttt{header}, \texttt{footer}\}.$$

Dimensions $v.h$ and $v.w$ in Oversized are the node's rendered bounding-box height and width in CSS pixels, obtained from the browser's layout engine.

**Algorithm 1** DIVIDEPAGE

**Input:** DOM root node $r$
**Output:** PageMem $p$

$L \leftarrow [\,]$
SPLIT($r$, $L$)
$p \leftarrow$ NEWPAGEMEM( )
$p.u \leftarrow$ CURRENTURL( )
$p.n \leftarrow$ EXTRACTTITLE( )
$p.S \leftarrow L$
**return** $p$

**procedure** SPLIT(node $v$, list $L$)
  **if** ISTERMINAL($v$) **then**
    append MAKESECTION($v$) to $L$
  **else**
    $C \leftarrow$ GROUPSIBLINGS($v$.children)
    **for** $c \in C$ **do**
      SPLIT($c$, $L$)

**function** ISTERMINAL(node $v$)
  **return** $v$.isListSection $\vee$ $v$.tag $\in \mathcal{T}_{\text{group}}$ $\vee$ $\neg$ OVERSIZED($v$)

**function** OVERSIZED(node $v$)
  **return** $(v.h > 900 \wedge v.w > 320) \vee (v.h > 500 \wedge v.w > 800)$

**function** GROUPSIBLINGS(children $(c_1, \ldots, c_k)$)
  scan $(c_1, \ldots, c_k)$ for groups of consecutive siblings that share tag and class
  replace each group of length $\geq 4$ with a single list-section node containing the group
  **return** the resulting (shortened) sequence

### A.1.3 PAGEMEM UPDATE.

UPDATEPAGEMEM refreshes a PageMem to reflect the current live page state and is invoked at the start of every observation step. It is also the routine that populates a freshly divided PageMem with its initial elements and summaries.

UPDATESECTION (i) queries the browser for the section's current set of interactable elements, (ii) computes the added / removed / modified diff $\Delta = (\Delta^+, \Delta^-, \Delta^\sim)$ against the section's previous element list, (iii) re-summarizes if the section has no summary yet or if the structural change is large enough, and (iv) returns $\Delta$ so that the caller (an observation step or a workflow) can respond to partial state changes. This is the machinery behind, e.g., the dropdown workflow of §3.5, which reads the revealed options directly from $\Delta^+$. List sections are handled specially: their content is unbounded and repetitive, so instead of enumerating elements or tracking a diff at the list level, we always re-summarize from a screenshot and return an empty diff. Element-level tracking happens only on the per-item sub-sections, and only after list-item selection in the observation pipeline (§3.4).

**Element population.** GETELEMENTS produces the element list for a section via a three-step pipeline. A helper first resolves the section to a Playwright locator. The locator's descendants are then filtered by the clickable predicate ISCLICKABLE defined below. Finally, each surviving DOM node is passed to an Element constructor that reads its DOM attributes into $\alpha_e$.

**Clickable predicate.** A DOM node $v$ is considered interactable iff it passes a visibility-and-accessibility gate *and* satisfies at least one positive signal. The gate excludes nodes that are not rendered, carry the `disabled` attribute, or have `aria-hidden="true"`. The positive signals are any of: a tag in an interactable tag set, a DOM event-listener attribute in a listener set, an ARIA role in an

**Algorithm 2** UPDATEPAGEMEM and UPDATESECTION

**procedure** UPDATEPAGEMEM(PageMem $p$)
  **for** $s \in p.S$ **do**
    UPDATESECTION($s$)
  **if** $p.\sigma_p$ is undefined **then**
    $p.\sigma_p \leftarrow$ VLMSUMMARIZEPAGE($p$)

**procedure** UPDATESECTION(PageSection $s$)
  **if** $s$ is a list section **then** ▷ list sections only get summarized
    $s.\sigma_s \leftarrow$ VLMSUMMARIZESECTION($s$)
    **return** $(\emptyset, \emptyset, \emptyset)$
  $E_{\text{new}} \leftarrow$ GETELEMENTS($s$)
  $\Delta \leftarrow$ DIFF($s.E_s$, $E_{\text{new}}$) ▷ $\Delta^+$: added, $\Delta^-$: removed, $\Delta^\sim$: input-value changes
  $s.E_s \leftarrow E_{\text{new}}$
  **if** $s.\sigma_s$ is undefined **then**
    $s.\sigma_s \leftarrow$ VLMSUMMARIZESECTION($s$)
  **else if** $|\Delta^+| + |\Delta^-| \geq 3$ **then** ▷ re-summarize if $\geq 3$ elements added/removed
    $s.\sigma_s \leftarrow$ VLMSUMMARIZESECTION($s$)
  **return** $\Delta$

interactable role set, or a computed `cursor` style of `pointer`. Formally,

$$\text{ISCLICKABLE}(v) \equiv \big(v.\text{tag} \in \mathcal{T}_{\text{clk}} \vee v.\text{attrs} \cap \mathcal{L}_{\text{clk}} \neq \emptyset \vee v.\text{role} \in \mathcal{R}_{\text{clk}} \vee v.\text{cursor} = \texttt{pointer}\big) \wedge \text{ACCESSIBLE}(v),$$

with the sets

$$\begin{aligned}\mathcal{T}_{\text{clk}} &= \{\texttt{button}, \texttt{a}, \texttt{input}, \texttt{select}, \texttt{textarea}, \texttt{details}, \texttt{summary}, \texttt{option}\},\\ \mathcal{L}_{\text{clk}} &= \{\texttt{onclick}, \texttt{onmousedown}, \texttt{onmouseup}, \texttt{onkeydown}, \texttt{onkeyup}\},\\ \mathcal{R}_{\text{clk}} &= \{\texttt{button}, \texttt{link}, \texttt{menuitem}, \texttt{option}, \texttt{radio}, \texttt{checkbox}, \texttt{tab},\\ &\qquad \texttt{textbox}, \texttt{combobox}, \texttt{slider}, \texttt{spinbutton}, \texttt{search}, \texttt{searchbox}\}.\end{aligned}$$

These heuristics are adapted from BrowserUse (Müller & Žunic., 2024).

### A.2 EXPLORATION

Exploration (§3.3) is a deterministic depth-first traversal of a website's pages and clickable elements, deduplicated against the running set $E_w$ of all elements seen on the site, with state restored between element clicks by reloading the pre-click URL. EXPLOREPAGE (Algorithm 3) is the recursive driver that visits one page at a time; it delegates to ITERATEPAGE and EXPLOREELEMENT (Algorithm 4) for the element-level work. Exploration is launched per website by initializing an empty $\mathcal{M}_w$ and a URL-only PageMem stub at a chosen starting URL — the homepage in our experiments — and invoking EXPLOREPAGE at the configured maximum depth. The pseudocode omits the per-page element, per-website page, and time budgets, which act as additional early-return checks; their values are in Table 7.

**Page-level traversal.** EXPLOREPAGE takes a stub PageMem carrying a target URL and the remaining recursion depth. It deduplicates against URLs already in $\mathcal{M}_w$, navigates the browser to the page, runs DIVIDEPAGE on the freshly-loaded DOM to construct the full PageMem, and UPDATEPAGEMEM to populate elements and summaries. The newly-built PageMem is then registered in $\mathcal{M}_w$. If its section structure matches an existing template in $T_w$, the page is treated as a known-shape duplicate and not iterated, since further iteration would re-cover element behaviors already learned from the matching template; otherwise, if the page contains a list section or its `is_list_item` flag is set, the PageMem is added to $T_w$ as a new template. ITERATEPAGE is then called on the page, returning a list of stub PageMems for newly-discovered URLs, which the procedure recursively explores at depth $d - 1$.

**Template matching.** MATCHESTEMPLATE compares the candidate PageMem against each template in $T_w$. Two PageMems match when they have the same number of sections and each pair of corresponding sections is structurally equivalent under the DOM-derived attributes in $\alpha_s$ (tag, class, and other selector-defining attributes). Because DIVIDEPAGE is deterministic on the DOM, structurally equivalent pages reliably yield identical section sequences in practice, so exact structural equality is sufficient as a match criterion without needing a similarity threshold. Matching is checked only against $T_w$ rather than all of $P_w$, both for efficiency and because non-template pages are by definition idiosyncratic and not expected to recur.

**Algorithm 3** EXPLOREPAGE

**procedure** EXPLOREPAGE(PageMem stub $p$, depth $d$)
  **if** $p.u \in \mathcal{M}_w.P_w$ **then** ▷ URL already explored on this site
    **return**
  NAVIGATE($p.u$)
  Populate $p$ with sections, elements and summaries
  add $p$ to $\mathcal{M}_w.P_w$ keyed by $p.u$
  **if** MATCHESTEMPLATE($p$, $\mathcal{M}_w.T_w$) **then** ▷ already explored page with same structure
    **return**
  **else if** HASLISTSECTION($p$) $\vee$ $p$.is_list_item **then**
    add $p$ to $\mathcal{M}_w.T_w$
  $N \leftarrow$ ITERATEPAGE($p$) ▷ explore elements on page
  **if** $d = 0$ **or** budget exhausted **then return**
  **for** $p' \in N$ **do**
    EXPLOREPAGE($p'$, $d - 1$)

**Element-level traversal.** ITERATEPAGE walks the page's elements in document order. For elements not contained in any list section, it skips those already in the global element set $E_w$ and registers each new element in $E_w$. List sections are handled separately: rather than iterating every list item (which would redundantly re-cover elements with structurally identical neighbors), the procedure invokes ITERATELISTITEM, which iterates the elements contained in a single list-item container using the same per-element logic. Stubs returned from list-item exploration are tagged with `is_list_item` so that the recursive EXPLOREPAGE call can promote the resulting pages to templates.

EXPLOREELEMENT returns the newly-discovered URL stub(s) reached by clicking the element. It first applies a static skip filter (described below) that rules out elements unsafe or unhelpful to click. It then records the pre-click URL, clicks the element, and inspects the result. If the URL has not changed, the post-click diff in page state is computed. Any newly-revealed elements ($\Delta^+$) are recorded as the clicked element's `dropdown_elements` and recursively explored using the same EXPLOREELEMENT routine. If the URL has changed and points to a same-site page not yet in $\mathcal{M}_w$, a URL-only stub is created and returned. Finally, the browser is reloaded to the pre-click URL to restore page state for the next iteration.

**Skip filter.** SHOULDSKIP excludes four categories of elements before any click is issued: (i) off-site links, identified by an `href` pointing to a domain outside $w$; (ii) authentication links such as login and sign-up, identified by keyword matching against the link text and URL path; (iii) `tel:`, `mailto:`, and `javascript:print(. . . )` links, identified by the `href` scheme; and (iv) *modifier* buttons that could mutate persistent site state, identified either by the form attribute `type="submit"` or by keyword matching of the element's accessible text against destructive terms (`delete`, `remove`, `submit`, `save`, etc.).

### A.3 OBSERVATION PIPELINE

ANALYZEPAGE (Algorithm 5) implements detail extraction and summary synthesis for the selected sections $S_t$, routing list sections through SELECTLISTITEMS (Algorithm 6).

**Per-section detail extraction.** For each selected section, a helper FORMAT produces the *details string* consumed by the extraction LLM. For a normal section, the details string contains the section's

**Algorithm 4** ITERATEPAGE and EXPLOREELEMENT

**procedure** ITERATEPAGE(PageMem $p$)
  $N \leftarrow [\,]$
  **for** element $e$ in $p$ not contained in a list section **do**
    **if** per-page element budget exhausted **then break**
    **if** $e \in \mathcal{M}_w.E_w$ **then continue**
    add $e$ to $\mathcal{M}_w.E_w$
    $N \leftarrow N +$ EXPLOREELEMENT($e$)
  **for** list section $s \in p.S$ **do**
    $N_\ell \leftarrow$ ITERATELISTITEM($s$) ▷ explore elements in one list-item container
    **for** $p' \in N_\ell$ **do** $p'$.is_list_item $\leftarrow$ true
    $N \leftarrow N + N_\ell$
  **return** $N$

**procedure** EXPLOREELEMENT(Element $e$)
  $N \leftarrow [\,]$
  **if** SHOULDSKIP($e$) **then return** $N$
  $u_{\text{pre}} \leftarrow$ CURRENTURL( )
  CLICK($e$)
  $u_{\text{post}} \leftarrow$ CURRENTURL( )
  **if** $u_{\text{post}} = u_{\text{pre}}$ **then**
    Identify newly revealed elements $\Delta^+$
    **if** $\Delta^+ \neq \emptyset$ **then** ▷ click revealed new elements (dropdown opened)
      $e$.dropdown_elements $\leftarrow \Delta^+$
      **for** $e' \in \Delta^+$ **do**
        $N \leftarrow N +$ EXPLOREELEMENT($e'$) ▷ explore dropdown elements
  **else if** $u_{\text{post}}$ is on the same site $w$ **and** $u_{\text{post}} \notin \mathcal{M}_w.P_w$ **then**
    create stub $p_{\text{new}}$ with $p_{\text{new}}.u \leftarrow u_{\text{post}}$
    $N \leftarrow N + [p_{\text{new}}]$
  NAVIGATE($u_{\text{pre}}$) ▷ restore pre-click state for the next iteration
  **return** $N$

accessibility subtree together with the URLs and VLM-generated descriptions of any images in the section above the minimum size threshold (50 x 50 pixels). For a list section, SELECTLISTITEMS is invoked first to choose a subset of items, and the details string is the per-item content formatted as a numbered list, with each entry containing the same accessibility-subtree-plus-image content as a normal section. The extraction call LLMEXTRACTDETAILS (Prompt D.1.1) caches its output on the PageSection together with the details string used to produce it; on a subsequent call with an identical details string, the cached extraction is returned without an LLM call. The page-level task summary, in contrast, is regenerated on every call to ANALYZEPAGE, since the relevant framing of a page can shift as the task history $h_t$ grows, whereas section summaries $\sigma_s$ persist across tasks in the WebsiteMem.

**Algorithm 5** ANALYZEPAGE

**procedure** ANALYZEPAGE(PageMem $p$, selected sections $S_t$)
  $X \leftarrow [\,]$ ▷ per-section extraction strings, in selection order
  **for** $s \in S_t$ **do**
    **if** $s$ is a list section **then**
      SELECTLISTITEMS($s$) ▷ populates $s$'s sub-sections and $s.E_s$
    $D \leftarrow$ FORMAT($s$)
    $x \leftarrow$ LLMEXTRACTDETAILS($D$) ▷ returns cached value if $D$ unchanged
    append `"<idx> <tag> <class>: "` $+ x$ to $X$
  $p$.task_summary $\leftarrow$ LLMSUMMARIZEPAGE($X$) ▷ regenerated every call
  **return** $p$.task_summary

**List item selection.** A list section can contain hundreds or thousands of items, far exceeding what fits in a single LLM context. SELECTLISTITEMS addresses this by chunking the items sequentially into fixed-size groups and prompting the LLM to select relevant items chunk by chunk (Prompt D.1.2). After each chunk, a separate LLM call is issued (Prompt D.1.2) that sees the indices already searched, the items already selected, and the remaining entries, and decides whether to terminate early — this avoids paying the cost of scanning the full list when the relevant items have already been found (e.g., the top few results of a sorted list). After selection, the procedure rebuilds the list section's sub-sections from the selected items, populating each via UPDATESECTION, and overwrites the list section's element list $E_s$ with only the elements from selected items. The original full element set is not retained: re-selection on a later observation step rebuilds the sub-sections from scratch from the live page state.

**Algorithm 6** SELECTLISTITEMS

**procedure** SELECTLISTITEMS(list section $s$)
  $I^\star \leftarrow [\,]$ ▷ indices of items selected so far
  partition the items of $s$ into sequential chunks $(C_1, C_2, \ldots)$ of fixed size $c$
  **for** $k = 1, 2, \ldots$ **do**
    $I^\star \leftarrow I^\star +$ LLMSELECTITEMS($C_k$, $I^\star$)
    **if** LLMCHECKDONE($I^\star$, indices searched so far, remaining items) **then**
      **break**
  rebuild $s.S'_s$ as one PageSection per item index in $I^\star$
  **for** $s' \in s.S'_s$ **do** UPDATESECTION($s'$)
  $s.E_s \leftarrow$ concatenation of $s'.E_{s'}$ over $s' \in s.S'_s$ ▷ only selected items contribute actions

### A.4 AGENT LOOP

AGENTLOOP (Algorithm 7) executes one timestep at a time until either the task is verified complete or the step budget is exhausted. Each timestep produces *one* agent action, which may itself be a compound workflow that internally issues multiple LLM sub-calls and browser operations. Bookmark and (where applicable) website pre-selection happen once at task start (Prompt D.2.1).

**Observation phase.** At the start of each timestep, the agent retrieves or constructs the PageMem $p_t$ for the current page and refreshes it via UPDATEPAGEMEM. A modal-detection helper CHECKMODAL then tests for the presence of a modal dialog using DOM heuristics including `role="dialog"` and `aria-modal="true"`; when a modal is detected, section selection is bypassed entirely and the modal's PageSection is used as the sole relevant section, focusing the agent's attention on the dialog and preventing the surrounding (now-inert) page from polluting the candidate space. Otherwise, the LLM selects relevant sections $S_t$ from the section summaries as in §3.4 (Prompt D.1.1). The remaining sections $S_t^{\complement} = p_t.S \setminus S_t$ are kept aside for use during candidate assembly. ANALYZEPAGE then produces the task summary $\hat{o}_t$.

**Action phase.** GATHERCANDIDATES assembles the candidate set $\mathcal{A}_t$ in two passes. First, for each $s \in S_t$, an LLM call selects elements from $s$ likely to be useful for the task, conditioned on $s$'s extracted details (Prompt D.2.1). Second, a single LLM call covers the elements of the first five entries of $S_t^{\complement}$ in document order — a heuristic that ensures upper-page UI (navigation bars, search boxes, primary buttons) remains reachable even when the LLM did not flag those sections as relevant during section selection. Navigation actions (visited URLs, bookmarks $B_\tau$, type-URL, switch-tab, switch-website) are filtered by an LLM pass against the task (Prompt D.2.1). The end-task action is always appended. A rule-based pre-filter removes irrelevant actions (e.g., switching to an already active tab, clicking an already-selected radio button, print, tel links, links leading outside allowed domains) before $\mathcal{A}_t$ is presented to the LLM.

The action-selection LLM call (Prompt D.2.1) returns a chosen action $a$ together with a natural-language *reason* $r$ explaining the choice. EXECUTEACTION dispatches to the workflow appropriate to the selected action's type. Navigation actions invoke a single NAVIGATE call; element actions inside a form section invoke SUBMITFORM, while other element actions invoke ELEMENTACTION, which routes to the appropriate workflow (App. A.5); the end-task action invokes LLMVERIFYENDTASK.

**Algorithm 7** AGENTLOOP

**procedure** AGENTLOOP(task $\tau = (I, u_0)$, WebsiteMem $\mathcal{M}_w$)
  $B_\tau \leftarrow$ LLMSELECTBOOKMARKS($\mathcal{M}_w$, $I$) ▷ optional, at task start only
  $h \leftarrow [\,]$
  **for** $t = 1, \ldots, T_{\max}$ **do**
    $p \leftarrow$ GETPAGEMEM(current URL, $\mathcal{M}_w$)
    **if** $m \leftarrow$ CHECKMODAL($p$) **then**
      $S \leftarrow [m]$ ▷ modal: focus on dialog, skip section selection
    **else**
      $S \leftarrow$ LLMSELECTSECTIONS($p$)
    $\hat{o} \leftarrow$ ANALYZEPAGE($p$, $S$)
    $u_{\text{pre}} \leftarrow$ current URL
    **for** $j = 1, \ldots, J_{\max}$ **do** ▷ intra-step continuation, $J_{\max} = 5$
      $\mathcal{A} \leftarrow$ GATHERCANDIDATES($p$, $S$, $p.S \setminus S$, $B_\tau$)
      $(a, r) \leftarrow$ LLMSELECTACTION($\mathcal{A}$) ▷ up to 3 retries on action error
      **if** $a$ is end-task **then**
        **if** LLMVERIFYENDTASK($I$, $h$) **then** ▷ 1 verification check per task
          **return** LLMFINALANSWER($I$, $h$)
        **else**
          remove end-task from $\mathcal{A}$ and re-prompt for $a$
      EXECUTEACTION($a$) ▷ single op or compound workflow
      **if** current URL $\neq u_{\text{pre}}$ **then break**
      **if** CHECKMODAL($p$) $\neq$ **nil then break**
      UPDATEPAGEMEM($p$)
      $\hat{o}' \leftarrow \hat{o} +$ VLMSCREENDIFF($p$) ▷ cheap update for follow-up action
    append step observation, reason, and action to $h$
  **return** LLMFINALANSWER($I$, $h$) ▷ step budget exhausted

If the chosen action raises a runtime error (an invalid URL, a stale selector, an interaction failure on a non-interactable element), the action is removed from $\mathcal{A}_t$ and the LLM is re-prompted; this retry budget resets each timestep and is bounded at three attempts.

**End-task verification.** When the LLM selects end-task, LLMVERIFYENDTASK issues an LLM call conditioned on the task instruction $I$ and the interaction history $h_t$ that judges whether the task has been completed (Prompt D.2.1). If completion is verified, a separate LLMFINALANSWER call (Prompt D.2.1) produces the answer string and the loop terminates. If not, the end-task action is removed from $\mathcal{A}_t$ for the current timestep only — it remains available on subsequent timesteps — and the LLM is re-prompted to choose a different action. We allow at most one verification check per task episode; subsequent end-task actions end the task immediately.

**Intra-step continuation.** Some actions (entering input into a field, copying to clipboard) leave the page on the same URL and complete a partial rather than full intent. To avoid the overhead of restarting the observation pipeline for follow-up actions on the same page, after such an action the loop enters a short continuation phase: UPDATEPAGEMEM refreshes the page state, a VLM is prompted to describe the visual difference and the description is concatenated with $\hat{o}_t$ to produce the updated observation $\hat{o}'_t$, and the LLM selects another action from a freshly-gathered candidate set. The continuation phase ends when either the page URL changes, a modal dialog appears (handled at the next timestep with focused attention), the agent selects end-task, or a budget of five within-step actions is reached.

### A.5 ACTION WORKFLOWS

This section details the element-action workflows dispatched by EXECUTEACTION (App. A.4).

**Element-type dispatch.** ELEMENTACTION (Algorithm 8) is a flat decision tree over element properties: it checks first for behaviors known from exploration (recorded `dropdown_elements`),

then for input-type-specific handlers (file upload, select/combobox, the various `input`/`textarea` subtypes), then for "probably opens something" signals (`aria-haspopup`, or any element not yet explored), and finally falls through to a plain click.

**Algorithm 8** ELEMENTACTION

**procedure** ELEMENTACTION(Element $e$)
  **if** $e$.dropdown_elements $\neq \emptyset$ **then** ▷ recorded from exploration
    DROPDOWNACTION($e$)
  **else if** $e$.input_type $=$ `file` **then**
    UPLOADFILE($e$)
  **else if** $e$.tag $=$ `select` $\vee$ $e$.role $=$ `combobox` **then**
    SELECTOPTION($e$)
  **else if** $e$.tag $\in \{$`input`, `textarea`$\}$ $\vee$ $e$.role $=$ `spinbutton` **then**
    **if** $e$.input_type $\in \{$`submit`, `reset`, `button`$\}$ **then**
      CLICKELEMENT($e$)
    **else if** $e$.input_type $=$ `search` **then**
      SEARCH($e$)
    **else if** $e$ is a radio or checkbox **then**
      CLICKELEMENT($e$)
    **else**
      ENTERINPUT($e$)
  **else if** $e$.aria-haspopup $\neq$ `false` $\vee$ $\neg\, e$.explored **then**
    DROPDOWNACTION($e$) ▷ probe for dropdown semantics
  **else if** $e$ is a copy button **then**
    COPYTOCLIPBOARD($e$)
  **else**
    CLICKELEMENT($e$)

**Form submission.** SUBMITFORM (Algorithm 9) handles forms in three phases. An LLM call first selects which fields to fill (Prompt D.2.2), and ELEMENTACTION is invoked on each chosen field, dispatching to ENTERINPUT, SELECTOPTION, or UPLOADFILE as appropriate. A validation pass then re-fills any field that is empty-but-required or carries `aria-invalid="true"` after the initial entry — the LLM is re-prompted for new values for these fields. Finally, a review loop allows the LLM to inspect the populated form and either edit additional fields, submit, or exit (Prompt D.2.2); exiting leaves the form in its current state and returns control to the main agent loop without submitting. Within the review loop the candidate set is restricted to elements of the form section.

**Algorithm 9** SUBMITFORM

**procedure** SUBMITFORM(form section $s$)
  $F \leftarrow$ LLMSELECTFIELDS($s$)
  **for** $f \in F$ **do**
    ELEMENTACTION($f$)
    UPDATESECTION($s$)
  **for** $f \in s.E_s$ where $f$ is empty-and-required **or** $f$.aria-invalid $=$ `true` **do**
    ELEMENTACTION($f$)
  UPDATESECTION($s$)
  **for** $k = 1, \ldots, K_{\max}$ **do** ▷ review phase, $K_{\max} = 15$
    $a \leftarrow$ LLMSELECTFORMACTION($s$) ▷ element in $s$, submit, or exit
    **if** $a$ is exit **or** $a$ is a submit button **then**
      **if** $a$ is a submit button **then** CLICKELEMENT($a$)
      **return**
    ELEMENTACTION($a$)
    UPDATESECTION($s$)
    **if** current URL has changed **then return**

**Dropdown action.** DROPDOWNACTION (Algorithm 10) clicks the dropdown trigger and consults the section diff returned by UPDATEPAGEMEM. Three outcomes are possible. If the URL changed or no new elements were revealed, the click was an ordinary navigation or null action and the workflow returns. If the revealed elements form a coherent form-like cluster (multiple inputs together with a submit-like button), control is routed to SUBMITFORM on the synthesized form section. Otherwise — the typical case of a menu, autocomplete list, or option dropdown — the LLM selects one of the revealed elements (Prompt D.2.1) and that element is clicked. The form-detection heuristic ISFORM returns true when $\Delta^+$ contains at least two input-like elements and at least one element matching submit-button heuristics (a button with type `submit`, or accessible text matching submit-like keywords).

**Algorithm 10** DROPDOWNACTION

**procedure** DROPDOWNACTION(element $e$)
  $u_{\text{pre}} \leftarrow$ current URL
  CLICKELEMENT($e$)
  $\Delta \leftarrow$ UPDATEPAGEMEM(current page)
  **if** current URL $\neq u_{\text{pre}}$ **or** $\Delta^+ = \emptyset$ **then return**
  **if** ISFORM($\Delta^+$) **then**
    SUBMITFORM(synthesize form section from $\Delta^+$)
  **else**
    $e' \leftarrow$ LLMSELECTACTION($\Delta^+$)
    CLICKELEMENT($e'$)

**Other workflows.**

- SEARCH: invokes ENTERINPUT on the search field, then computes a section diff to detect whether suggestions have appeared. If they have, the LLM is offered the option to select a suggestion (which is then clicked) or to ignore them (Prompt D.2.2). The workflow concludes by pressing the Enter key to issue the search.
- ENTERINPUT: prompts the LLM for the value to enter (Prompt D.2.2) and fills it into the field.
- UPLOADFILE: presents the LLM with the choice of either an existing file in the agent's local filesystem (input files staged for the task — e.g., the input images supplied with VisualWebArena tasks — and any text files created earlier in the same task or VLM-captioned images saved during the task) or a *create-new-file* option (Prompt D.2.2). In the latter case the agent is prompted for a filename and text content (Prompt D.2.2), the file is written to the local filesystem, and the new file is then uploaded.
- SELECTOPTION: prompts the LLM to choose one of the available options (Prompt D.2.2) and sets the field's value to that option via Playwright.
- COPYTOCLIPBOARD: reads the copied text from the clipboard and logs it in the action history.
- CLICKELEMENT: issues a Playwright click on the element.

### A.5.1 ACTION LOGGING AND HISTORY FORMAT

Every successful basic action contributes a string to the agent's interaction history. Failed actions (those that raise the runtime errors handled by the retry mechanism in App. A.4) are not logged — only the eventually-successful action appears. Action strings are constructed *after* execution, since several formats reference values that are known only post-execution (e.g., the actual text entered into a field, the option that was selected). Table 4 lists the format for each basic action.

Compound actions and intra-step continuation chains produce multiple basic-action strings within a single timestep. These are emitted as a list under an `Actions:` heading in the history; a timestep that produced exactly one basic action uses the singular `Action:` heading instead. Each timestep contributes a block of the form below to the history, with the page name and URL drawn from the PageMem, the task summary from ANALYZEPAGE, and the reason produced by the LLM in the same call as the action selection itself.

| Action | Log string format |
|---|---|
| CLICKELEMENT | `Clicked {elem_name}` |
| COPYTOCLIPBOARD | `Copied text to clipboard: "{value}"` |
| SELECTOPTION | `Selected value of "{option}" for field "{elem_name}"` |
| ENTERINPUT | `Entered value of "{input_value}" into {elem_name}` |
| UPLOADFILE | `Uploaded file: "{file}" into the "{elem_name}" field` |
| SWITCHTAB | `Switched from tab {start_i} ({start_url}) to tab {new_i} ({new_url})` |
| SWITCHWEBSITE | `Navigated to the "{website_name}" website ({website_url})` |
| GOTOPAGE | `Navigated to page "{page_name}" ({page_url})` |
| GOBACK | `Returned to the "{page_name}" page` |
| ENTERURL | `Navigated to URL: "{url}"` |

Table 4: Log string format for each basic action.

```
* Step 1:
  * Observation: {page_name} ({url})
    - Summary: {task_summary}
  * Reason for Action: {reason}
  * Action: {action_str}
* Step 2:
  * Observation: {page_name} ({url})
    - Summary: {task_summary}
  * Reason for Action: {reason}
  * Actions:
    - {action_str}
    - {action_str}
```

# B ADDITIONAL EXPERIMENT DETAILS

## B.1 FULL COMPARISON INCLUDING PROPRIETARY AND LARGE-MODEL AGENTS

Table 5 extends Figure 2 with every published agent we are aware of, including those built on proprietary or >120B open-weight models; the main text omits them because their per-task cost is one to two orders of magnitude higher than that of the small models we study, and because a small model in the WebChallenger harness already matches or exceeds them (§4.1).

**URL replacement on WebArena and VisualWebArena.** WebArena and VisualWebArena evaluate against locally-hosted simulated copies of real websites (Reddit, GitLab, OpenStreetMap, etc.), but task instructions refer to these sites by their real names. We observed that LLMs frequently misinterpret this as a directive to navigate to the real site — e.g., reading the localhost URL, concluding "I am not on Reddit," and attempting to navigate to `https://www.reddit.com`, which breaks evaluation. We address this confusion with a bidirectional URL substitution applied at the prompt boundary: simulated-site URLs are rewritten to their real-site counterparts in every string passed to the LLM, and the inverse rewrite is applied to URLs in LLM outputs before they reach the browser. The LLM thus reasons consistently as if it were operating on the real site, while the browser remains pointed at the simulation. Table 6 lists the seven substitutions used.

**Multi-website selection (WebArena).** At the start of each task, the agent is presented with the full list of benchmark websites and prompted to select any sites beyond the starting URL that are relevant (Prompt D.3). The homepages of selected sites are added to the bookmark set $B_\tau$, making them available as one-click navigation actions throughout the task.

**Input image grounding (VisualWebArena).** A subset of VisualWebArena tasks include input images that the agent must reason over alongside the task instruction. At task start, the VLM is prompted with the task instruction, the input image(s), and the current page screenshot, and asked to produce a textual description of the image(s) in relation to the task (Prompt D.3). This description is appended to the task instruction for the duration of the task.

Table 5: Full comparison, including agents built on proprietary or >120B open-weight models, which cost one to two orders of magnitude more per task than the small models in §4.1 and are listed here for reference only. Success rate (%) on four benchmarks. Small denotes open-weight models with ≤120B parameters. Within the proprietary block and within the small open-weight block, the best and second-best value in each column are shaded dark and light green. GenericAgent scores are from the BrowserGym leaderboard (ServiceNow, 2025). WA: WebArena, VWA: VisualWebArena, OMW: Online-Mind2Web, WoA: WorkArena L1.

| Method | Model(s) | WA | VWA | OMW | WoA |
|---|---|---|---|---|---|
| *Proprietary and large open-weight models* | | | | | |
| GenericAgent | GPT-4o | 31.4 | 26.7 | – | 45.5 |
| GenericAgent | Claude 3.5 Sonnet | 36.2 | 21.0 | – | 56.4 |
| GenericAgent | GPT-5 | – | – | – | 79.1 |
| GenericAgent | GPT-4o-mini | 17.4 | 16.9 | – | 27.0 |
| WALT (Prabhu et al., 2025) | GPT-5 | 50.1 | 52.9 | – | – |
| WebTactix (WebArena Team, 2026) | DeepSeek-V3.2 | 74.3 | – | – | – |
| IBM CUGA (Marreed et al., 2025) | – | 61.7 | – | – | – |
| OpenAI CUA (OpenAI, 2025) | – | 58.1 | – | 61.3 | – |
| ScribeAgent (Shen et al., 2024) | GPT-4o + Qwen2.5-32B | 53.0 | – | – | – |
| AgentSymbiotic (Zhang et al., 2025b) | Claude 3.5 + Llama3 8B | 48.5 | – | – | – |
| AgentOccam-Judge (Yang et al., 2025) | GPT-4 | 45.7 | – | – | – |
| WebPilot (Zhang et al., 2024) | GPT-4o | 37.2 | – | – | – |
| Agent Workflow Memory (Wang et al., 2024) | GPT-4o | 35.5 | – | – | – |
| SkillWeaver (Zheng et al., 2025a) | GPT-4o | 29.8 | – | – | – |
| *Small open-weight models (fine-tuned)* | | | | | |
| Agent-as-Annotators (Lù & Reddy, 2026) | Qwen3.5-9B | 41.5 | 33.9 | – | 51.5 |
| Mobile-Agent-v3.5 (Xu et al., 2026) | Qwen3-VL-32B | 48.4 | 46.6 | – | – |
| WebDreamer (Gu et al., 2025) | Qwen2-VL-7B | – | 21.9 | 35.0 | – |
| Fara-7B (Awadallah et al., 2025) | Qwen2.5-VL-7B | – | – | 34.1 | – |
| Learn-by-Interact (Su et al., 2025) | Codestral 22B | 24.2 | – | – | – |
| AgentTrek (Xu et al., 2025) | Qwen2.5-32B | 22.4 | – | – | – |
| Go-Browse (Gandhi & Neubig, 2025) | Qwen2.5-7B | 21.7 | – | – | – |
| AutoWebGLM (Lai et al., 2024) | ChatGLM3 6B | 18.2 | – | – | – |
| TTI (Shen et al., 2025) | Gemma 3 12B | 26.1 | – | – | – |
| *Small open-weight models (zero-shot)* | | | | | |
| GenericAgent | GPT-oss-120b | – | – | – | 50.9 |
| Tree Search (Koh et al., 2025) | Llama-3-70B-Instruct | 10.1 | 16.7 | – | – |
| **WebChallenger (Ours)** | GLM-4-32B + Qwen2.5-VL-7B | **56.3** | **48.7**† | **51.0** | **70.9** |
| **WebChallenger (Ours)** | Qwen3.5-9B | 47.1 | **58.3** | – | **83.3** |
| **WebChallenger (Ours)** | GPT-oss-120b | – | – | – | **77.4** |

† Our VisualWebArena experiments use Qwen3-VL-4B-Instruct in place of Qwen2.5-VL-7B-Instruct.

| Simulated host | Real-site URL |
|---|---|
| `$SHOPPING` | `https://www.onestopmarket.com` |
| `$CLASSIFIEDS` | `https://osclass-classifieds.com` |
| `$REDDIT` | `https://www.reddit.com` |
| `$WIKIPEDIA` | `https://en.wikipedia.org` |
| `$SHOPPING_ADMIN` | `http://magento.site/` |
| `$GITLAB` | `https://gitlab.com` |
| `$MAP` | `http://openstreetmap.org` |

Table 6: URL substitutions applied at the prompt boundary on WebArena and VisualWebArena. Environment variables hold the localhost URLs of the simulated sites. The substitution is applied bidirectionally: simulated → real on input to the LLM, real → simulated on URLs in the LLM's outputs.

**Hyperparameters.** Table 7 consolidates the configuration values used across all components of the system. These hyperparameters were largely chosen heuristically as reasonable defaults and were not extensively swept, as we did not observe strong sensitivity in pilot runs. When the per-website exploration timeout is reached, exploration stops and the partial WebsiteMem is used as-is. All runs are single-seed at temperature 0 (Appendix E.5).

| Component | Parameter | Value |
|---|---|---|
| Exploration | max elements per page | 75 |
| | max pages per website | 500 |
| | max depth (homepage at depth 0) | 2 (1 for Online-Mind2Web) |
| | max time per website | 12h (1h for Online-Mind2Web) |
| Page division | oversize thresholds $(h, w)$ | $(>900, >320)$ or $(>500, >800)$ |
| | list-grouping run length | $\geq 4$ |
| Section update | resummarization threshold $\lvert\Delta^+\rvert + \lvert\Delta^-\rvert$ | $\geq 3$ |
| Observation | list-item chunk size $c$ | 25 |
| | minimum image size for VLM description | $50 \times 50$ px |
| Agent loop | max steps $T_{\max}$ | 30 |
| | intra-step continuation budget $J_{\max}$ | 5 |
| | action-error retries per step | 3 |
| | end-task verification attempts per task | 1 |
| Form submission | review loop bound $K_{\max}$ | 15 |

Table 7: Consolidated hyperparameter values. Per-benchmark exploration overrides are noted parenthetically; all other values are identical across the four benchmarks.

### B.2 WebTactix Harness Comparison

We evaluate WebTactix (WebTactix) and WebChallenger on the same 165 WebArena-lite tasks (Table 2), with two backbones each. Both use the official WebArena evaluator. Maps tasks use live OpenStreetMap, with reference answers checked against that interface; the same task definitions and references apply to all four conditions. We reset mutable benchmark sites between conditions and run five workers, serializing tasks that share a mutable site.

All four conditions use low reasoning effort and a 16,384-token output limit. WebTactix uses 15 reasoning rounds with up to four parallel branches and a 10-row table limit. Both WebChallenger backbones use the same configuration: 30 agent steps and a 50-row table limit. Its observation pipeline summarizes sections in one batched call and extracts details from selected sections, together with the overall page summary, in another. Section selection and per-section action candidate selection remain separate; memory and compound workflows are retained. The fixed exploration memory uses the original Qwen2.5-VL summaries.

### B.3 Compute Resources and Exploration Coverage

The main four-benchmark experiments ran locally using vLLM (Kwon et al., 2023) on a desktop with one NVIDIA RTX 3090 GPU, an AMD Ryzen 5 3600 CPU, and 64 GB RAM. We report offline memory construction separately from task execution because the resulting memories are reused across tasks.

**Offline exploration.** The six WebArena environments and WorkArena were explored concurrently; Online-Mind2Web was explored in batches of 6–10 concurrent websites. VisualWebArena reused memories for its shared WebArena environments and added a Classifieds memory.

Table 8: Offline exploration coverage and summarization usage. Token counts include VLM input and output; shared memories are counted only once.

| **Memory scope** | **Stored pages** | **Summarization tokens** |
|---|---:|---:|
| WebArena | 787 | ∼16M |
| WorkArena | 55 | ∼7M |
| Online-Mind2Web | 1,154 | ∼27M |
| VisualWebArena: additional Classifieds memory | 45 | Not recorded |

WebArena's stored pages comprise Shopping (400), Reddit (166), Shopping Admin (128), Maps (8), GitLab (82), and Wikipedia (3). Counts refer to URL-keyed PageMem records in the frozen exploration memories, excluding separately stored list templates. Subdomains and redirects can

produce multiple hosts for one Online-Mind2Web website, while blocked sites may have no stored pages.

**Task execution.** Tasks used approximately 270k tokens on average across the controller and vision model; we report token usage rather than run time, since the latter depends on serving throughput rather than on the harness (Appendix E.5).

### B.4 Per-Site Ablation Results

Table 9 breaks the component ablations of Table 3 down by WebArena-lite site. With 21 to 46 tasks per site, one task is 2–5 points and one standard error 7–11 points, so per-site differences below about ten points, including the one-task gain of the memory ablation on Maps, are within noise.

Table 9: Component ablations on WebArena-lite by site (success rate, %). Per-site task counts: Shopping ($n = 46$), Reddit ($n = 21$), GitLab ($n = 32$), Maps ($n = 31$), CMS ($n = 35$).

| Method | Shop. | Reddit | GitLab | Maps | CMS | Avg | Δ |
|---|---|---|---|---|---|---|---|
| *WebChallenger* | 56.5 | 71.4 | 65.6 | 45.2 | 60.0 | 58.8 | – |
| − memory | 43.5 | 71.4 | 53.1 | 48.4 | 48.6 | 50.9 | −7.9 |
| − compound actions | 47.8 | 66.7 | 50.0 | 48.4 | 40.0 | 49.1 | −9.7 |
| − observation pipeline | 50.0 | 57.1 | 53.1 | 25.8 | 25.7 | 41.8 | −17.0 |

## C Broader Impacts

Capable web agents on small, locally runnable open-weight models make web automation cheaper, keep sensitive browsing sessions on the user's device, and make agent research accessible to groups without large compute budgets. The same accessibility lowers the barrier for misuse such as spam posting, fake account creation, and review manipulation. Because errors compound over long autonomous horizons and robustness to adversarial pages is uncharacterized (§5), we recommend human oversight in any consequential domain.

## D Prompts

### D.1 Observation Prompts

#### D.1.1 Section Selection, Extraction, and Summary Prompts

**SelectSections**

```
You are an intelligent virtual assistant who completes tasks for users
on various websites. You will be given information about your task, the
previous steps taken, and a list of summaries for the different sections
available on the current page. You must now identify all page sections
that seem relevant to the task based on their descriptions and select those
sections for further analysis.

The information will be given as:
TASK: task request from the user
HISTORY: summary of previous task steps
CURRENT PAGE: name and url of current page
PAGE SECTIONS: list of sections available on the current page

Think step-by-step and identify all page sections that might contain
relevant details or potentially aid in progressing the task. Then, select
the relevant page sections by providing a list of integers corresponding
to the indices of your selected sections.
```

```
Give your full thought process behind your reasoning as:
*THOUGHTS** your reasoning steps

Give your answer for the relevant sections as a comma-separated sequence
of integers corresponding to your choices:
*RELEVANT SECTIONS**: indices of selected page sections (e.g., "2, 5, 3")

Additional Guidelines:
- Select all sections that seem like they could potentially be useful at
any point in the task.
- If you think a section might be relevant but are not sure, select it to
see its full details.
```

**ExtractDetails**

```
You are an intelligent virtual assistant who completes tasks for users
on various websites.  To complete your task, you must first analyze
each section of the page one-by-one and identify any relevant details.
You will be given information about your overall task and the next
section of the page to analyze. You must then summarize any information or
elements in the page section that might potentially be relevant for the task.

The following information will be provided:
TASK: your task instructions
HISTORY: summary of previous task steps
CURRENT PAGE: title, url, and summary of current page
SECTION: page section to analyze
CONTENT: section content

First,  think  step-by-step  about  the  current  subsection  of  the  page
you are analyzing and consider whether more information needs to be
gathered. Then, provide a summary of the information and save the relevant
details in this format: `- **{descriptive_label}**: "{value}"`

Give your full thought process followed by your answer in this format:
*THOUGHT**: your reasoning
*SUMMARY**: summary of why the information may or may not be relevant to
the task
*RELEVANT DETAILS**: noteworthy information from this section

Additional guidelines:
- Do not include instructions for the next steps to take just yet, only
highlight information or potentially useful actions from the current page.
- Never make assertive statements or assumptions about the given page
section as it only provides a partial view of the website. Always mention
any potential uncertainties and/or alternative possibilities about the
information given.
- Make sure to mention any page details that could help us find the correct
path to take even if the information is not directly relevant itself.
- Your response should end after you provide the relevant details.
```

**SummarizePage**

```
You are a virtual AI assistant who performs tasks on behalf of users through
a web browser. You will be provided with your assigned task instructions,
the previous interaction history, and the current web page observation.
You must now write a concise summary of the current page that highlights
```

```
the relevant information for your task.

The following information will be provided:
TASK: your task instructions
HISTORY: previous task steps
CURRENT PAGE: title and url of current page
PAGE OBSERVATION: overview of page

First, think step-by-step about the progress you have made previously in
order to understand the context of the current situation. Then, identify
the information on the page that is the most relevant to your task and
provide a concise report (max one paragraph in length) that summarizes the
current observation.

Provide your full thought process followed by your summary of the
page observation in this format:
*THOUGHT**: your reasoning steps
*OBSERVATION SUMMARY**: summary of relevant details

Additional Guidelines:
- You should not decide what action to take yet, only highlight relevant
information.
- Do not make assumptions about the functionality of the elements on the
page. You may describe what you think will be a likely result of clicking
an action, but you should mention any potential uncertainties and not make
strong statements.
- Make sure to include in your summary any details from the current page
that would be worth remembering in the future.
- If the page is not relevant to your task then simply provide a brief
description of the page and why it is not relevant.
- Your answer should end after you provide the observation summary.
```

#### D.1.2 List Item Selection Prompts

**SelectItems**

```
You are an intelligent virtual assistant who completes tasks for users on
various websites. You will be requested by the user to complete a task
that involves finding one or more items in a list of content (e.g. a list
of links, products, articles, comments, etc.) on the website. The list of
items on the current page will be shown and you will have the option to
select any items that are relevant to the task based on their positional
indices.

The following information will be provided:
TASK: the user's task instructions
HISTORY: summary of previous task steps
PAGE: title and url of the current page
LIST INFO: summary describing the content list
LIST ITEMS: the list items on the current page of results

Think step-by-step about the given information and determine if there are
any relevant list items on the current page of results. Then, select
all items that are relevant to the task in any way by providing a
comma-separated list of integers corresponding to the positional indices
of those items (e.g. "2, 9, 13"). If there are no exact matches in the
list, then still select the item that is the closest match. If all items
```

```
are completely unrelated to the task, then you can answer 'None'.

Provide the full thought process behind your answer followed by your
final answer for the item selection in this format:

*THOUGHTS**: your reasoning steps

*SELECT ITEMS**: list items to select (list of integers or 'None')

Additional Guidelines:
- If some of the list items seem like potential matches but there is not
enough information to confirm, select them in order to view more details.
- Select all items that are partially related to the task requirements even
if they are not exact matches.
- Always include all items that are mentioned in the task instructions.
- End your response after selecting the list items.
```

**CheckDone**

```
You are an intelligent virtual assistant who completes tasks for users
on various websites. Your current objective requires you to find one or
more items in a list of content (e.g. a list of pages, products, articles,
comments, etc.). In the previous steps you have iterated over the items
in order and recorded any matching items found. You will be given the
history of the items you have checked so far and you must determine if
the objective is complete or if you still need to find more items in the list.

You will be provided with the following information:
PAGE: title, url, and description of the current page
OBJECTIVE: the task you must complete
ITEMS CHECKED: list of items that have been checked so far, and whether
they match

Think step-by-step about the requirements of the objective and the
details about the items that have been checked so far. Then, determine if
the task is complete or if you still need to look for more items in the
rest of the list.

Give your full thought process and your answer in this format:
*THOUGHTS**: steps behind your reasoning process
*COMPLETE**: whether the objective has been completed (Yes/No)

Reminder:
- Pay attention to the sort order of the list. It may allow you to determine
if the remaining list items are worth checking (e.g. if you need to find
the cheapest item and the list is sorted by price, then the remaining items
will be more expensive).
- If it is still possible that there might be more matching items in the
list, then you should keep searching to be sure.
```

### D.2 Action Prompts

#### D.2.1 Agent Loop Prompts

**SelectBookmarks**

```
You are a virtual AI assistant who performs tasks on behalf of users through a web browser. You will be given a user task request and a list of pages on the current website that you can visit in order to carry out the task. You must now identify any pages that are likely to be relevant for completing the assigned task.

The list of available pages will be given in this format:
ALL PAGES:
1) {page_1}
2) {page_2}
...
n) {page_n}

You should carefully think step-by-step to determine whether any of the pages would likely be useful for completing the task. Include all pages that have the potential to be helpful for completing the task in any way.
Provide your reasoning as:
*THOUGHTS**: your reasoning steps

Finally, select the relevant pages by providing a comma-separated sequence of integers corresponding to the indices of your choices:
*RELEVANT PAGES**: indices of selected pages(s) (e.g. "1, 7")

Note:
- Select any pages that seem like they might be partially relevant or help find the target page quicker.
- If there are no pages that seem relevant for the task then provide 'None' as your answer.
```

**ElementCandidates**

```
You are a virtual AI assistant who performs tasks on behalf of users through a web browser.  You will be provided with a list of clickable elements available on the current page and you must identify all candidate elements that are potentially useful for progressing the task.

Your assigned task and the previous interaction history will be provided as:
TASK: task instructions
HISTORY: summary of previous task steps
CURRENT PAGE: title, url, and summary of current page observation

The elements present on the page will be listed in this format:
PAGE ELEMENTS:
1) {element_1}
2) {element_2}
...

You should carefully think step-by-step to determine which elements might help progress the task. Provide your full reasoning process as:
*THOUGHTS**: your reasoning steps
```

```
Finally, select one or more elements by providing the indices of
your choices separated by commas:
*SELECT ELEMENTS**: indices of selected element(s) (e.g. "1, 7")

Note:
- For search/filtering interfaces, make sure to clear all pre-existing
filters before applying a new one.
- If there are multiple elements that seem relevant and you are not sure
which one is best, then include all relevant elements in your answer.
- If none of the listed elements are relevant then answer 'None'.
```

**NavigationCandidates**

```
You are a virtual AI assistant who performs tasks on behalf of users through
a web browser. You will be provided with your assigned task instructions,
the previous interaction history, current page observation, and a list of
browser navigation actions to choose from. You must now determine if any
of the available navigation actions would be useful for progressing the task.

The following information will be provided:
TASK: your task instructions
HISTORY: summary of previous task steps
CURRENT PAGE: title, url, and summary of current page
BROWSER ACTIONS: list of available navigation options and their details

Think step-by-step about the navigation options and whether they would
be helpful or harmful for progressing the task. Then, select one or more
candidates for the next action to take. If none of the navigation options
are useful for the task then select 'None'.

Give your full thought process behind your reasoning as:
*THOUGHTS** your reasoning steps

Select one or more of the navigation options by providing a comma-separated
sequence of integers corresponding to the indices of your choices:
*ANSWER**: indices of selected option(s) (e.g. "1, 7")
```

**SelectAction**

```
You are an autonomous AI assistant who performs tasks on behalf of users
through a web browser. You will be provided with your assigned task
instructions, the previous interaction history, current page observation,
and a list of valid actions to choose from. You must now determine the
correct next action to take in order to progress with the task, or end the
task if it has been completed.

The following information will be provided:
TASK: your task instructions
HISTORY: summary of previous task steps
CURRENT PAGE: title, url, and summary of current page observation
ACTIONS: list of available action options

You should first carefully think step-by-step about the potential
benefits and risks of each option in order to determine the logical next
action for making progress on your task. Provide your full thinking process
as:
*THOUGHTS**: your reasoning steps
```

```
Once you have decided on the next action you wish to take, provide
the reason behind your decision followed by the index of your selected
action in the following format:
*REASON**: reason for decision
*SELECT ACTION**: index of selected action (e.g. "1")

Note:
- The reason in your final answer should be a one sentence summary of the
thinking process behind your decision.
- You must select exactly ONE action out of the options provided.
- Do not mark the task as complete until you have fully completed all steps
specified by the instructions (e.g. if the task is to buy a product, you
need to complete the full checkout and payment process).
- Only provide the index of the selected action with no additional text
afterwards.
```

**VerifyEndTask**

```
You are an intelligent virtual assistant who completes tasks for users on
various websites. Based on your given task instructions, your previous
action history and the current page observation, you must determine whether
you have fully completed the task.

The following information will be provided:
TASK: task instructions from the user
HISTORY: summary of previous task steps
CURRENT PAGE: name, url and summary of the current browser page
ACTIONS: available actions (not executed yet)

Think step-by-step about user's task instructions and identify which
parts of the task you have accomplished, then determine whether the task
is fully complete or additional actions need to be taken.

*Guidelines for evaluation**:
1. You are only required to complete the steps explicitly defined by the
instructions.
2. As long as there are any next steps that need to be taken, the task
completion status should be false.
3. Tasks that involve buying an item should only be considered complete
after adding the item to the cart and completing the full checkout process.
4. If the user asks you to show them an item (e.g. "show me the most
expensive ...", "show me the most recent ...", "find me a ..." etc.), then
the task is not complete until the full details of the item are displayed
by clicking its link.
5. If the task instructions asks you to open a page (e.g. "open my latest
..."), then the task is not fully complete until you have clicked the link
for the page.

Provide your thought process followed by your answer in this format:
*THOUGHTS**: your reasoning process
*TASK COMPLETE**: task completion status (True/False)
```

**FinalAnswer**

```
You are an intelligent virtual assistant who performs tasks for users on
various websites. Previously, you have completed the user's task on the
```

```
website and now you must send a final message to the user.

You will be provided with the following information:
USER: the task request from the user
HISTORY: summary of previous task steps
CURRENT PAGE: title, url, and summary of current page

Think step-by-step the user's task instructions and the information
you have found on the webstite. Then, provide a message to the user that
summarizes the steps you have taken to complete their task followed by the
final answer to the user's query.

Outline your full thought process as:
*THOUGHTS**: your reasoning

Provide your message to the user followed by your answer value in
this format:
*MESSAGE**: completion message
*ANSWER**: final answer value

Important guidelines for the final answer:
- If you were unable to successfully complete the task or no answer is
necessary, provide "N/A" as the answer.
- When asked to return a count, return the count as a number with units
instead of "N/A" if it's 0.
- If the user asks you to check if something is true or not, answer "Yes"
or "No".
```

#### D.2.2 Action Workflow Prompts

**SelectFields**

```
You are an intelligent virtual assistant who completes tasks for users on
various websites. You will be requested by the user to complete a task that
involves filling in a web form. The list of available form fields will be
provided and you will be able to select which ones to edit based on the task.

The task instructions from the user and your current subgoal will
be provided as:
TASK: the task assigned by the user
HISTORY: the previous interaction history
CURRENT PAGE: name, url and summary the of current page

The form will be provided as a numbered list of available input
fields:
FORM:
1) {field}
2) {field}
...
n) {field}

Think carefully about the task and identify the input fields that
you need to select, then provide your reasoning as:
*THOUGHT**: reasoning

Finally, provide the list the of all form fields that should be
updated for the task by providing a comma-separated sequence of integers
```

```
corresponding to the indices of your choices as:
*EDIT FIELDS**: list of indices (e.g. "1, 7")
```

**SelectFormAction**

```
You are an intelligent virtual assistant who completes online form filling
tasks on various websites. In the previous steps, you filled out the
available fields of the form and you will now choose the next action to
take in the form.

The task information will be given in the following format:
TASK: overall task to complete
PROGRESS: the current task progress
FORM MENU: numbered list of actions available in the form section

Think step-by-step about the given information and identitify the
appropriate next action to take among those listed in the form menu. You
will then be able to select one of the actions by providing its index.

Provide your full thought process followed by your action choice
as:
*THOUGHTS**: your reasoning steps
*CHOICE**: index of selected action (e.g. "**CHOICE**: 14")

Additional Guidelines:
- Don't repeat the action performed in the previous step.
- End your response after providing your action choice.
```

**EnterInputValue**

```
You are an intelligent virtual assistant who completes tasks for users on
various websites. You will be provided information about your assigned
task, the web page, and the current state of the task progress. You have
just selected an input field on the page and you can now provide a value
to enter into the input field.

The task information will be provided in this format:

TASK: <user's task instructions>

PAGE: <summary of page>

PROGRESS: <current task progress>

INPUT FIELD: <selected input field>

Think step-by-step about the actions needed to complete the task and
then answer with the value to enter into the selected input field. If the
task specifies an exact input value to enter then your answer should match.
Otherwise, come up with a reasonable input value to enter given the task
context.

Provide your full thought process followed by your final answer for
the input value as:

*THOUGHTS**: <reasoning>
```

```
*INPUT VALUE**: <value>

Additional Guidelines:
- If the task does not directly specify an input value to use then think
carefully about what would be the best value to enter in the field.
- Only provide an input value for the current selected input field with no
additional text afterwards.
```

**SelectSearchSuggestions**

```
You are an intelligent virtual assistant who completes tasks on various
websites. You will be provided information about the current page and your
assigned task. A search bar displaying a list of search suggestions is
currently selected and you will now be have the option to click one of them.

The information about your task, the current browser page and the
selected search bar will be provided as:
TASK: <your assigned task>
PAGE: <summary of page>
SEARCH FIELD: <details about the search bar>
SUGGESTIONS: <list of search suggestions>

Think step-by-step about whether any of the search suggestions are
relevant to the task. If you want to click one of the suggestions then
provide the number and value of the option. If there are no relevant search
suggestions then you can choose the 'None' option. If the correct input
value is already entered into the field and the same value also appears in
the search suggestions, you should still select the same value again.

Provide your full thought process followed by your final answer in
this format:
*THOUGHTS**: <reasoning process>
*SELECT**: <number and value of option> (e.g. "1) Meta")
```

**SelectOption**

```
You are an intelligent virtual assistant who completes tasks on various
websites. You will be provided information about the current page and your
assigned task. You have selected an input field on the page with several
options available and you must select the appropriate numbered option from
the list by providing its index.

The information about the page, your task, and the options for the
input field will be provided in this format:

TASK: task instructions

PROGRESS: current task progress

PAGE: summary of page

INPUT FIELD: selected input element

INPUT OPTIONS:
1) {option_1}
2) {option_2}
```

```
...
n) {option_n}

Think step-by-step about the actions needed to complete the task and
then answer with the value to enter into the selected input field. If the
task specifies an exact input option to select then your answer should
match. Otherwise, choose the option that most closely matches with the
task instructions given the task context.

Provide your full thought process followed by your final answer as:

*THOUGHTS**: your reasoning

*OPTION**: index of selected option (e.g. "1")

Additional Guidelines:
- Only select ONE of the input options.
- Don't provide any additional text after the selected option.
```

**UploadFile**

```
You are an intelligent virtual assistant who completes tasks for users
on various websites. Your current task involves uploading a file and
you have already navigated to the part of the website where the file
upload should take place. The list of files available in your filesystem
will be given and you must choose now choose the appropriate file to upload.

The following information will be provided:
TASK: your task instructions
HISTORY: summary of the previous actions you have taken
CURRENT PAGE: title, url, and summary of current page
INPUT FIELD: the field that you are uploading the file to
FILES: list of files to choose from

Based on this information, you should carefully analyze the context
of your current task and determine which file from your filesystem
should be uploaded. If the task requires creating a new file to upload,
then you can select the 'Create new file' option to write and upload a file.

Think step-by-step and provide your full thought process as:
*THOUGHTS**: your reasoning steps

Then, choose one of the options by providing the index of your choice as:
*ANSWER**: chosen index (e.g. "1")
```

**CreateFile**

```
You are an intelligent virtual assistant who completes tasks for users
on various websites. Your current task involves creating a new file and
uploading it to the current website. Based on the given task context,
you must create the file that will be uploaded by providing the name and
contents of the file.

The following information will be provided:
TASK: your task instructions
HISTORY: summary of the previous actions you have taken
CURRENT PAGE: title, url, and summary of current page
```

FIELD: the field where the file will be uploaded

Reason carefully about your task and plan what you will name the file and what the file contents should be. Once you are ready, you can provide the file name followed by the full file content and this file will be uploaded to the website.

Provide your full thinking process as:
*THOUGHTS**: your reasoning steps

Provide the name of the file followed the file content in this format:

*FILE**: file name (e.g. urls.txt)

*CONTENT**:
“‘
# full file content here ...
“‘

**EnterURL**

You are an autonomous virtual assistant who completes tasks for users on various websites by controlling a web browser. You will be provided information about your assigned task, the previous actions you have taken, and the current page information. The browser address bar has been selected and you must now provide a URL to enter and navigate to.

You will be given the following information:
TASK: your task instructions
HISTORY: summary of previous task steps
CURRENT PAGE: title, url, and summary of current page observation

Think carefully about the given task information and determine the correct URL for the page that you wish to navigate to on the website.

Important guidelines:
1. The URL must be for the same website as the current page.
2. Pay close attention to the previous URLs you have visited in order to help understand the URL structure used by the website.

Provide your full reasoning process followed by your answer for the URL in this format:
*THOUGHTS**: your reasoning steps
*ANSWER**: URL

### D.3 Benchmark-Specific Prompts

**Multi-site (WebArena)**

You are an autonomous AI assistant who performs tasks on behalf of users through a web browser. You will be given a task from the user and start on a website where the task needs to be completed on. First you must determine if the task only involves the current website, or if part of the task needs to be completed on another website.

You will be given the following information:
USER: task instructions from the user

```
CURRENT PAGE: description of current browser page
WEBSITES: list of available websites

Carefully read the user's instructions and reason about whether the
task involves only the current website, or if a task requirements involves
another website. In most cases the task will only focus on the current site
but occasionally it will require information or functionality from one of
the other websites. Your final answer should include one or more websites
that are necessary for the task and must always include the current website.

Provide your reasoning process followed by your final answer as:
*THOUGHTS**: your reasoning steps
*ANSWER**: website(s) needed for the task (e.g "OneStopMarket", "Reddit +
GitLab")
```

**InputImage (VisualWebArena)**

```
Describe what the following user is asking for based on their request
message that references both the first image and the web page screenshot.

*User request message**: "{task_instruction}"
```

### D.4 VLM Prompts

**VLMSummarizeSection**

```
Describe this section of the {website_name} website in one sentence.
```

**VLMSummarizePage**

```
Describe this page of the {website_name} website in one sentence.
```

**VLMScreenDiff**

```
These two screenshots show the {website_name} website immediately before
and after I {action_str}. What was the result of the action?
```

**DescImageShort**

```
Describe the image with a short sentence.
```

**DescImageLong**

```
Fully extract all information contained in this image in an organized format.

Make sure to include:
- All textual and/or numerical information.
- Visual features such as objects, colors, and settings.
- Names of any recognizable people, celebrities, characters, media, or
locations.
```

# E CLAIMS, BOUNDS, COMPARATORS, AND REPRODUCIBILITY

This appendix states what the system guarantees by construction and what it guarantees only empirically, explains the choice of comparators and configurations in §4, and collects the details needed to reproduce the numbers in the paper.

## E.1 WHAT "CONSTRUCTED BY FIXED RULES" MEANS

PageMem has two kinds of content. Its *structure*, the section tree, the list-section grouping, the element sets, and the CSS selectors, is a pure function of the DOM and the rendered layout (Algorithms 1 and 2) and involves no model call; given the same DOM and viewport it is identical across runs. Its *summaries*, one sentence per section and per page, are written by a model and are therefore only as reproducible as deterministic decoding of that model. Everywhere the paper says PageMem is built by fixed rules, the claim is about the structure; summaries are cached (Appendix E.3) so that a given section is summarized once per website rather than once per visit.

## E.2 PER-CALL INPUT BOUND

The observation pipeline never places a whole page in one prompt: every detail-extraction call receives the content of at most one PageSection, and every list-selection call receives at most $c = 25$ list items. A PageSection produced by DIVIDEPAGE is one of (i) a non-grouping DOM node whose rendered box is not oversized, i.e. whose height is at most 900 or width at most 320 pixels, and whose height is at most 500 or width at most 800 pixels (Table 7); (ii) a list section, presented $c$ items at a time; or (iii) a node whose tag is a grouping tag. Cases (i) and (ii) are bounded by construction, (i) by the rendered area, since text density per pixel is bounded on real pages, and (ii) by $c$. Case (iii) is the one place where a single node can be large, for example a `form` or `table` without repeated children; siblings sharing tag and class inside such a node are still merged into a list section by GROUPSIBLINGS, which covers most large tables and forms. The section-selection call sees one summary per section, about 20 tokens each, so its length grows with the number of sections rather than with their content. The bound is therefore structural rather than a hard token cap, and its practical effect is visible in Table 3: on WebArena-lite the mean input of an LLM call is 1,850 tokens with the pipeline against 8,793 for the single-prompt baseline, whose prompt grows with the whole page.

## E.3 CACHE SOUNDNESS

Three caches are used at inference: per-section task extractions $x$, task-independent section summaries $\sigma_s$, and the page-level task summary $\hat{o}_t$. With deterministic decoding, which we use for all locally served models, each behaves as follows.

1. **Extractions are exact.** LLMEXTRACTDETAILS caches its output $x$ together with the details string $D$ it was computed from, and returns $x$ only if the new details string equals $D$ exactly. A fresh call on the same $D$ would return the same $x$, so a cache hit never differs from recomputation.
2. **Section summaries are task-independent.** $\sigma_s$ is a general description of a section (Prompt VLMSUMMARIZESECTION) that does not mention the task or the history, so no change of task or subgoal can make it stale. It can become stale only when the section's content changes; UPDATESECTION recomputes the section's element set at every step and re-summarizes when at least three elements are added or removed, so staleness is confined to changes of fewer than three elements or to text-only changes.
3. **The task summary is never cached.** $\hat{o}_t$ is regenerated at every step from the current extractions and the current history $h_t$ (Algorithm 5).

Two consequences matter for correctness. First, a subgoal change cannot surface stale facts: the facts the agent reasons over come from extractions, which are recomputed whenever the section content changes, and are re-framed for the current subgoal by the task summary at every step. Second, a stale section summary can only affect *which* sections are selected, never *what* is read from them, since extraction always reads live content; its failure mode is a missed section, which the five-section fallback in candidate assembly (Appendix A.4) partially covers. The extraction cache is keyed on content but not on history, so an extraction computed under an earlier history is reused while

the content is unchanged; this is deliberate, since extractions record task-relevant content and the history-dependent framing is left to the task summary. API models add provider-side nondeterminism that these caches do not remove; the local runs behind Table 3 do not use API models.

### E.4 Ablations That Isolate One Component at a Time

The component ablations in Table 3 each remove more than one thing at once (§5). The following runs, all on WebArena-lite with GLM-4-32B, separate the confounded factors; Table 10 reports them.

1. **Observation vs. history.** Remove the observation pipeline but keep the summarized-observation history, so that only the observation prompt changes; the result (41.8%) equals the full observation ablation of Table 3, so the whole $-17.0$ is due to reading the page whole and none to the history format.
2. **Memory decomposed.** Two runs that remove only bookmarks or only the recorded dropdown contents; removing the dropdown contents alone reproduces the full memory ablation (50.9%), so they carry the memory effect, while bookmarks alone are worth 3.6 points, within one standard error.
3. **Observation stages.** Section selection without detail extraction, where the selected sections' raw content is passed to the summary call, costs 7.3 points, the marginal value of the extraction stage.
4. **Action interface.** The numbered-list action interface of §3.5 against native tool calling with the same candidate set: tool calling makes fewer LLM calls per task (95 against 154) but loses 6.7 points, which substantiates the reliability claim made there.

Table 10: Single-factor ablations on WebArena-lite (165 tasks, GLM-4-32B). $\Delta$ is relative to the full system (58.8%); one standard error is about four points. Calls and tokens are GLM-4-32B calls and tokens per task; the full-system values are derived from the run totals in Table 3.

| Variant | Success (%) | $\Delta$ | Calls/task | Tokens/task |
|---|---|---|---|---|
| *WebChallenger* (full) | 58.8 | – | 154 | 285k |
| − observation pipeline, history kept | 41.8 | $-17.0$ | 64 | 200k |
| − bookmarks only | 55.2 | $-3.6$ | 156 | 292k |
| − dropdown contents only | 50.9 | $-7.9$ | 145 | 276k |
| − detail extraction (selection only) | 51.5 | $-7.3$ | 134 | 252k |
| Tool-calling action interface | 52.1 | $-6.7$ | 95 | 158k |

### E.5 Choice of Comparators and Configurations

This section records why each comparison in §4 is set up as it is, and what each does not show.

**Why Claude Code and Codex.** The frontier comparison in §4.3 is a matched-backbone comparison, like Table 2: the question is what a harness adds to a given model, so the natural comparator for a frontier model is the harness its vendor ships with it and reports its agentic results through. Claude Code and Codex are general-purpose agents rather than web-specific ones, and drive a real Chrome instance through their tool interface; that generality is the setting a practitioner faces when handing a frontier model a browser, and it is the same setting in which these models' browsing abilities are advertised. Codex with Gemini 3.8 Flash doubles as the control a reviewer may ask for, the same VLM in a general harness without page decomposition: WebChallenger+ with the same model matches its success on both benchmarks at 4–4.4$\times$ lower cost, and the original WebChallenger pipeline with the same model does so at 1.6–2$\times$ lower cost. Deployed open-source browser harnesses such as Browser Use, Stagehand, and Skyvern report results with backbones and task sets that differ from ours, so a published-number comparison would not be backbone-matched; running them on our subsets with the same four backbones is the obvious next experiment and was outside our API budget for this submission.

**Why GenericAgent, WebTactix, and these backbones.** GenericAgent is the reference agent of BrowserGym, the platform on which WebArena, VisualWebArena, and WorkArena are run, so it is the standard single-prompt harness for a fixed model; we run it with our own GLM-4-32B so that

the only difference from WebChallenger is the harness. WebTactix was the top-ranked agent on the official WebArena leaderboard at the time of evaluation; its leaderboard entry uses DeepSeek-V3.2 on the full benchmark, which we could not afford to run for both harnesses, so the comparison in Table 2 is on WebArena-lite with two inexpensive API models, GLM-5.3-Flash and Gemini 3.8 Flash, at equal settings, and we make no claim about the leaderboard itself. GPT-4o-mini and GPT-5 in the lower block of Table 2 are a cheap and a strong API model, chosen to bracket GLM-4-32B and show that the harness neither depends on nor is tuned to one model.

**Headline configuration.** The GLM-4-32B configuration is the one evaluated on all four benchmarks and in every ablation, and was fixed before the base models in Figure 2 were released; the harness-versus-fine-tuning runs use each baseline's own base model and were run only on the benchmark the baseline reports. That Qwen3.5-9B, a newer model, exceeds the GLM configuration on VisualWebArena (58.3 vs. 48.7) and WorkArena L1 (83.3 vs. 70.9) is a further instance of the harness benefiting from stronger models (Table 2), not an inconsistency; both configurations are reported.

**Tokens rather than time.** We report no wall-clock times anywhere in the paper. Latency is set by serving throughput, provider traffic, prompt-cache hits, output length, and the number of sequential round trips, none of which is a property of the harness alone, and it is not proportional to tokens; the same harness on the same model can run at very different speeds on different days or endpoints. On a fixed model, inference cost is proportional to the tokens processed, so tokens and cost are the quantities on which harnesses can be compared; Table 15 gives the per-task token accounting behind every run in §4.3, and Appendix B.3 the token usage of the four-benchmark evaluation.

**Variance.** All runs are single-seed at temperature 0; for locally served models decoding is deterministic, and the remaining sources of variation are page-load timing and, for API models, provider-side batching. A full WebArena run costs about 270k tokens per task over 812 tasks on the single GPU used for the main results, so we report standard errors from the binomial estimate instead of repeated runs: about four points on WebArena-lite, 1.7–1.8 points on the full WebArena and VisualWebArena, and four to six points on the 40- and 45-task subsets of §4.3, and we flag every difference in the text that falls within one standard error.

**Budgets and adapters in the frontier comparison.** WebChallenger+ is capped at 100 tool-selection cycles or 60 minutes per task; Claude Code and Codex run uncapped. The cap can only lower WebChallenger+'s success, and an uncapped agent can spend more on tasks it fails, which raises its mean cost; on WorkArena L3 the uncapped agents fail at most four of 40 tasks, so this effect is bounded by a few tasks. Codex with Gemini 3.8 Flash runs through an API-compatibility adapter that retains Codex's Chrome tools and per-task sessions; it is not a vendor-supported configuration and is included because it is the only way to hold that backbone fixed across harnesses.

### E.6 Reproducibility Details

Code is released at: https://anonymous.4open.science/r/BUA-Webchallenger-6A2C/README.md

**Model serving and decoding.** The main experiments serve `GLM-4-32B-0414` and the vision models with vLLM (Kwon et al., 2023) on one NVIDIA RTX 3090 (24 GB) with greedy decoding (temperature 0). API models (GPT-5, GPT-4o-mini, GLM-5.3-Flash, Gemini 3.8 Flash) use the provider defaults except for temperature 0, low reasoning effort, and a 16,384-token output limit. All runs are single-seed (Appendix E.5).

**Human evaluation on Online-Mind2Web.** Following the benchmark's protocol, each of the 300 trajectories was judged against the benchmark's per-task rubric from the task, the final page, and the full action history, and marked as success, failure, or infeasible; the judged trajectories will be released with the code so that the judgments can be audited or re-scored with the benchmark's automatic judge.

**State resets.** WebArena, VisualWebArena, and WorkArena environments are reset to their reference snapshot between runs, and within a run tasks that mutate a shared site are serialized (Appendix B.2).

Agent memory is reset to the post-exploration WebsiteMem before every task, so no information leaks between tasks.

**Task manifests.** WebArena-lite is the 165-task subset of Liu et al. (2024); the WebChallenger+ subsets are listed in Tables 11 and 12; all manifests, per-task outcomes, and token logs will be released with the code.

**Benchmark adaptations and their provenance.** Two adaptations are shared by every system we ran, ours and baselines alike: the WebArena Maps environment is replaced by public OpenStreetMap, with the reference answers of seven map tasks re-derived against the live map, and the fuzzy-match evaluator uses GPT-4 Turbo in place of the original model; the modified task files are released with the code. Published baseline numbers in Table 5 and Figure 2 are taken from the cited papers or leaderboards without re-running and use the unmodified environments; the map change touches seven of 812 tasks and the evaluator change affects only tasks scored by fuzzy matching, so neither can account for the differences reported.

## F WEBCHALLENGER+ DETAILS

**Task selection.** We select representative subsets covering the main task categories and procedure families in WorkArena L3, and the websites, task types and difficulty levels in WebArena. Tables 11 and 12 list the selected tasks. All WorkArena episodes use task seed 0; the WebArena subset contains 45 distinct intent templates, including 14 hard tasks.

**Execution and evaluation.** Tasks sharing a mutable website are run sequentially to avoid interference. Tasks are prepared with authenticated browser access where required, and agents receive no task-specific assistance during execution. WebChallenger+ controls headless Chrome through its browser tools; Claude Code and Codex use their respective Chrome interfaces. All systems submit completion, answers or infeasibility as text. WorkArena uses its validator to check ordered progress during execution and completion at the end. Agents receive general instructions for common operations; Claude Code and Codex additionally receive instructions to wait 30 seconds after record changes or applied filters and before ending on the final page.

WebArena uses the original evaluator with two environment adaptations shared across systems: public OpenStreetMap with adjusted reference answers for seven map tasks, and GPT-4 Turbo for fuzzy answer matching.

**Model configurations.** WebChallenger+ uses each backbone at the same reasoning setting as its native harness: Gemini 3.8 Flash at the provider's default, Fable 5.1 at medium, and GPT-6 Astra at low. On both benchmarks, Codex uses GPT-5.6 Sol and GPT-6 Astra at low reasoning, and Gemini 3.8 Flash through an API compatibility adapter that retains its Chrome tools and per-task sessions. Claude Code uses Fable 5.1 at medium reasoning. Medium for Fable 5.1 in Claude Code and low for the OpenAI models in Codex are their respective default reasoning settings.

**Observation and interaction.** WebChallenger+ presents an outline of page regions with headings or short text previews, the presence of interactive controls and their location relative to the viewport. The outline has at most 50 entries, with labels of approximately 110 characters or fewer drawn directly from page content. The agent can scroll to a region, search page text or use visual grounding to locate a control. Later observations can include changed page content and reference context to reduce repeated input.

**History, memory and budget.** WebChallenger+ retains six recent interaction rounds in detail and summarizes older history in groups of twelve. This within-task history is separate from cross-task site memory, which is disabled in this comparison. Gemini prompt caching is configured with a five-minute lifetime. The budget is 100 tool-selection cycles or 60 minutes per task, followed by a short opportunity to submit a final answer. Claude Code and Codex are not subject to the 100-cycle limit.

Table 11: WorkArena L3 subset (40 tasks). Names are shown without the common `workarena.servicenow.` prefix and `-l3` suffix, with hyphens replaced by spaces. Each listed size is a separate task; a dash denotes a task without a size variant. All tasks use seed 0.

| Task | Size variants |
|---|---|
| Amount based expense management | small, medium, large |
| Basic expense management | small, medium, large |
| Basic filter problems and mark duplicates | medium |
| Dashboard retrieve catalog and max order developer laptop | – |
| Dashboard retrieve catalog and max order development laptop PC | – |
| Dashboard retrieve catalog and median order developer laptop | – |
| Dashboard retrieve incident and min filter asset list | – |
| Dashboard retrieve incident and mode create incident | – |
| Date based expense management | small, medium, large |
| Easy expense management | small, medium, large |
| Filter random expenses and find total return | small |
| Filter random expenses find total return and select investments | large |
| Filter requested items and order sales laptop | – |
| Filter single item expenses and find total return | small |
| Filter single item expenses find total return and select investments | medium |
| Get warranty expiration date | – |
| High priority filter problems and mark duplicates | large |
| Infeasible navigate and order Apple Watch with reason | – |
| Infeasible navigate and order developer laptop | – |
| Infeasible navigate and order development laptop PC | – |
| Infeasible navigate and order iPad mini | – |
| Infeasible navigate and order iPad pro | – |
| Infeasible navigate and sort hardware list | – |
| Navigate and filter service catalog item list | – |
| Off board user | – |
| On board user | – |
| Priority assignment | small, medium, large |
| Priority filter problems and mark duplicates | large |
| Three changes basic uniform risk change request scheduling | – |
| Three changes basic varied risk change request scheduling | – |

Table 12: WebArena subset: 45 task IDs from the original benchmark.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 23 | 49 | 66 | 75 | 87 | 92 | 94 | 97 | 105 |
| 196 | 210 | 213 | 235 | 255 | 258 | 344 | 353 | 357 |
| 367 | 386 | 403 | 440 | 451 | 469 | 483 | 490 | 525 |
| 540 | 543 | 574 | 588 | 598 | 605 | 656 | 674 | 681 |
| 695 | 705 | 718 | 734 | 736 | 766 | 788 | 789 | 800 |

Table 13: Numbers behind Figure 3: success and estimated inference cost per task on the WorkArena L3 and WebArena subsets. Low and medium denote each provider's default reasoning setting.

| | | WorkArena L3 (40) | | WebArena (45) | |
|---|---|---|---|---|---|
| **System** | **Backbone / effort** | **Success** | **USD/task** | **Success** | **USD/task** |
| *WebChallenger+* | Gemini 3.8 Flash | 38/40 (95.0%) | 0.21 | 35/45 (77.8%) | 0.11 |
| *WebChallenger+* | Fable 5.1 / medium | 39/40 (97.5%) | 1.21 | 36/45 (80.0%) | 0.42 |
| *WebChallenger+* | GPT-6 Astra / low | 40/40 (100.0%) | 1.48 | 36/45 (80.0%) | 0.65 |
| *WebChallenger* | Gemini 3.8 Flash | 38/40 (95.0%) | 0.53 | 34/45 (75.6%) | 0.25 |
| Claude Code | Fable 5.1 / medium | 36/40 (90.0%) | 1.64 | 38/45 (84.4%) | 1.14 |
| Codex | Gemini 3.8 Flash | 36/40 (90.0%) | 0.85 | 35/45 (77.8%) | 0.49 |
| Codex | GPT-5.6 Sol / low | 38/40 (95.0%) | 1.94 | 35/45 (77.8%) | 0.55 |
| Codex | GPT-6 Astra / low | 40/40 (100.0%) | 4.06 | 35/45 (77.8%) | 1.11 |

**Inference cost accounting.** We estimate inference cost from recorded token usage and the rates in Table 14. Averages include successful and failed tasks and exclude task setup, grading, orchestration, exploratory runs and report generation. Codex–Gemini accounting includes all agent attempts attributed to each task. WebChallenger+ accounting includes auxiliary grounding and history-summarization calls. Separately reported Gemini thinking tokens are charged as output; cached and uncached input are priced separately, and cache-storage costs are excluded. Claude's per-task estimates are rounded to cents.

Table 14: Recorded accounting rates in USD per million tokens. A dash indicates no separately modeled cache-write charge.

| Model | Input | Cache read | Cache write | Output |
|---|---|---|---|---|
| Gemini 3.8 Flash | 0.75 | 0.075 | – | 3.75 |
| Claude Fable 5.1 | 10.00 | 0.25 | 12.50 | 50.00 |
| GPT-5.6 Sol | 5.00 | 0.50 | – | 30.00 |
| GPT-6 Astra | 10.00 | 1.00 | – | 50.00 |

**Token accounting.** Table 15 reports, for each run behind Figure 3, the number of tool calls per task and where the input tokens go: the prompt tokens sent for the first time in a session, the size of the first model call, the uncached input tokens the provider billed, and the context size at the end of a task and at its peak. We report no wall-clock times, since latency depends on serving throughput, provider traffic, caching, and the number of sequential round trips rather than on the harness (Appendix E.5). The table shows where the cost gap comes from: WebChallenger+ keeps the peak context at 28–45k tokens and the first call at 9–17k, where Claude Code and Codex start every task with a 29–92k prompt and run at 70–190k tokens of context, so every subsequent call re-reads a much longer prefix.

Table 15: Token accounting for the WebChallenger+ comparison, per task. Tools: tool calls. New prompt: prompt tokens sent for the first time in the session, i.e. not served from the prompt cache, as counted by our logging; 1st call: prompt size of the first model call, dominated by the system prompt and tool definitions; fresh: uncached input tokens as billed by the provider. Final and peak context: model context size in tokens at the end of a task and at its maximum (mean/max). Backbones run at their native harness's default reasoning setting unless marked.

| | | | | | Input tokens / task | | | Context (mean/max) | |
|---|---|---|---|---|---|---|---|---|---|
| **System** | **Backbone** | **Success** | **USD** | **Tools** | **New** | **1st call** | **Fresh** | **Final** | **Peak** |
| *WorkArena L3 (40 tasks)* | | | | | | | | | |
| *WebChallenger+* | Gemini 3.8 Flash | 38/40 | 0.212 | 53.1 | 60.8k | 11.6k | 77.6k | 23.8k/37.9k | 40.9k/71.6k |
| *WebChallenger+* | Fable 5.1 | 39/40 | 1.210 | 31.4 | 41.6k | 17.1k | 50.5k | 32.8k/50.3k | 44.5k/54.1k |
| *WebChallenger+* | GPT-6 Astra | 40/40 | 1.480 | 35.2 | 37.5k | 9.3k | 56.8k | 21.6k/33.4k | 31.9k/40.7k |
| Claude Code | Fable 5.1 | 36/40 | 1.637 | 24.1 | 42.8k | 39.6k | 91.7k | 82.4k/113.9k | 82.4k/113.9k |
| Codex | Gemini 3.8 Flash | 36/40 | 0.852 | 43.0 | 127.4k | 92.2k | 509.1k | 161.0k/222.4k | 192.9k/245.9k |
| Codex | GPT-5.6 Sol | 38/40 | 1.942 | 30.4 | 83.2k | 36.5k | 95.1k | 119.7k/188.9k | 119.7k/188.9k |
| Codex | GPT-6 Astra | 40/40 | 4.060 | 30.9 | 68.6k | 54.8k | 77.7k | 123.4k/182.2k | 123.4k/182.2k |
| *WebArena (45 tasks)* | | | | | | | | | |
| *WebChallenger+* | Gemini 3.8 Flash | 35/45 | 0.111 | 29.6 | 37.1k | 11.6k | 35.9k | 24.6k/37.9k | 31.9k/47.0k |
| *WebChallenger+* | Fable 5.1 | 36/45 | 0.415 | 11.7 | 15.3k | 17.0k | 18.6k | 28.3k/46.0k | 30.4k/54.8k |
| *WebChallenger+* | GPT-6 Astra | 36/45 | 0.650 | 14.1 | 20.7k | 9.2k | 33.4k | 24.6k/44.1k | 27.7k/45.3k |
| Claude Code | Fable 5.1 | 38/45 | 1.144 | 21.2 | 24.0k | 44.5k | 61.8k | 68.5k/123.8k | 68.5k/123.8k |
| Codex | Gemini 3.8 Flash | 35/45 | 0.488 | 25.7 | 60.9k | 89.5k | 242.6k | 140.3k/223.9k | 142.2k/244.6k |
| Codex | GPT-5.6 Sol | 35/45 | 0.549 | 10.2 | 36.3k | 35.3k | 41.1k | 71.6k/148.1k | 71.6k/148.1k |
| Codex | GPT-6 Astra | 35/45 | 1.112 | 9.0 | 45.2k | 28.9k | 51.3k | 74.2k/122.3k | 74.2k/122.3k |
| Codex | GPT-6 Astra / medium | 34/45 | 1.148 | 9.1 | 45.9k | 29.6k | 52.0k | 75.6k/131.3k | 75.6k/131.3k |