WEBLICA: SCALABLE AND REPRODUCIBLE TRAINING ENVIRONMENTS FOR VISUAL WEB AGENTS

Anonymous authors
Paper under double-blind review

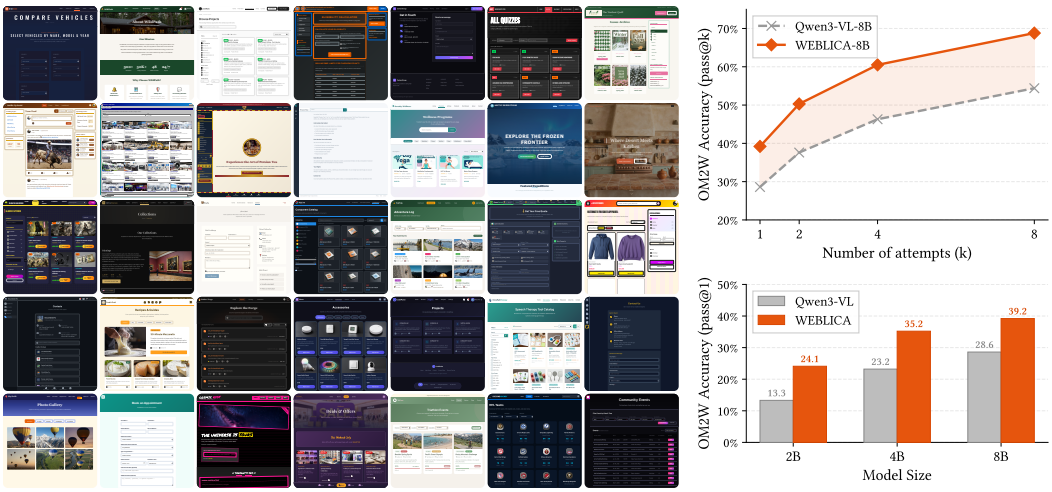


Figure 1: Scaling training data for visual web agents is challenging due to the complex, dynamic, and open-ended nature of the web. **Left:** We synthetically generate web environments in a fully automated and scalable manner, spanning a broad set of capabilities like navigation, form filling, filtering, date picking, and more. These provide a fully offline alternative to live-web training. **Right:** Training on these environments improves performance across multiple web navigation benchmarks (Online-Mind2Web (Xue et al., 2025) shown here), scaling with both test-time compute (top) and model size (bottom).

ABSTRACT

The web is complex, open-ended, and constantly changing, making it challenging to scale training environments for visual web agents. Existing data collection efforts remain limited to offline trajectories for supervised fine-tuning or a handful of simulated environments for reinforcement learning (RL), failing to capture the diversity of the web. Training directly on live websites could alleviate this, but it is slow, brittle, irreproducible, and can have unintended side effects on real websites.

We propose WEBLICA (Web Replica), a framework for constructing reproducible and scalable web environments. At its core, WEBLICA uses coding agents to synthesize interactive websites grounded in real-world websites and core web navigation skills, allowing us to scale RL training to thousands of diverse environments served locally. To assess how well synthetic websites can substitute for real ones, we further build a proxy for live-web training by caching real websites at the HTTP level, capturing and replaying stable visual states while preserving interactive behavior.

Both synthetic and cached real environments yield substantial improvements over the base model, with purely synthetic RL achieving the best overall performance. Our best model, WEBLICA-8B, outperforms open-weight baselines of similar size across multiple web navigation benchmarks while using fewer inference steps, scales favorably with additional test-time compute, and is competitive with API models.

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.

1 INTRODUCTION

Large language models are increasingly capable as autonomous agents in domains like coding (Anthropic, 2025a; OpenAI, 2025b; Wang et al., 2024; Hui et al., 2024), mathematics (Hubert et al., 2025; Novikov et al., 2025), and computer use (Anthropic, 2024; OpenAI; Google; Qin et al., 2025). This progress is driven by the availability of large-scale, high-quality training data and environments. Web navigation has emerged as a recent focus toward building agents that autonomously navigate the web to solve tasks ranging from information retrieval to form filling to online shopping. These agents aim to complete multi-step workflows that currently require manual effort, representing a significant step toward personalized digital assistants.

However, progress on building web agents has been slower, primarily due to the difficulty of scaling training data and environments to capture the complexity of the web. Recent attempts at data generation include collecting offline trajectories as demonstration data for supervised fine-tuning (Wang et al., 2025b; Awadallah et al., 2025; Gupta et al., 2026), offering limited support to handle the stochastic nature of the web due to lack of interaction. As an alternative, building simulated web environments offers interaction, but they commonly cover only a handful of manually defined domains (Zhou et al., 2023; Koh et al., 2024), limiting generalization. While training directly on live websites could alleviate that, it suffers from brittleness due to timeouts and bot detection, making training unstable, and exploratory actions can have unintended side effects on real websites. In addition, the live web is constantly evolving and slow to interact with, making carefully controlled ablations and fully reproducible training difficult. This raises a natural question: *how to scale interactive web environments while maintaining reproducibility?*

We propose WEBLICA, a framework for constructing reproducible and scalable web environments to train visual web agents. At its core, WEBLICA uses agentic coding tools to synthesize interactive websites grounded in real-world websites and common web navigation skills (e.g., form submission, authentication flows, dynamic search). Each environment is generated from a target navigation capability, a website category, and a visual style, and the coding agent iteratively checks the rendered site via screenshots and fixes it until the website and its tasks are functional. Because each environment is defined by its own code, it exposes its full interactive state, and its tasks are written together with the environment rather than limited to what an existing agent can already solve. To assess how well synthetic websites can substitute for real ones, we further build a proxy for live-web training via HTTP-level caching. Specifically, we record browsing sessions on thousands of real websites and replay them under complete network isolation, capturing stable visual states while preserving interactive behavior. All environments are served locally, eliminating network latency and enabling fast training.

Using this framework, we scale RL training to thousands of diverse environments and web navigation tasks. We fine-tune models from the Qwen3-VL (Yang et al., 2025a) family, which operate purely on screenshots without requiring set-of-marks annotations or DOM access, as these can hurt generalization due to the web's inconsistent underlying structure (Yutori, 2025). We find that training on either synthetic or cached real websites substantially improves performance over the base model, with purely synthetic RL performing better overall. These results suggest that synthesized websites can effectively substitute for real ones as RL environments. Our best model, WEBLICA-8B, reaches 39.2% pass@1 on Online-Mind2Web (Xue et al., 2025) with only 30 steps, outperforming open-weight models that use $3\times$ more steps, and improves further with additional test-time compute. WEBLICA also enables performance competitive with API models and consistent gains across model sizes, making synthesized environments a promising path to training visual web agents.

2 RELATED WORK

We review the most closely related work below and provide an extended discussion in Appendix A.

Building Web Agents. Early web agents relied on text-only language models that process structured representations such as accessibility trees or DOM elements (Yao et al., 2022a; Deng et al., 2023; Zhou et al., 2023; Gur et al., 2023). Later work adopted vision-language models (VLMs) to ground actions visually (Koh et al., 2024; He et al., 2024; Hong et al., 2024). Since early VLMs had limited grounding capabilities, initial approaches augmented screenshots with set-of-marks (Yang et al.,

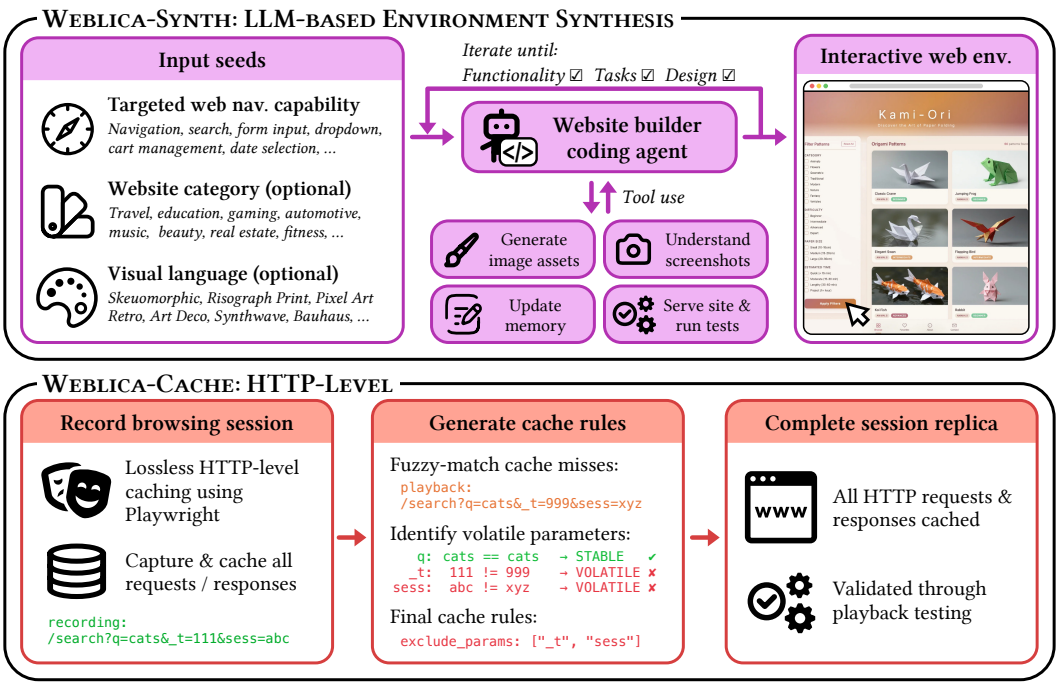


Figure 2: Framework overview. **WEBLICA-SYNTH (top)**: We task coding agents to generate web environments parameterized by a navigation capability, website category, and visual style. The agent writes framework-free HTML, CSS, and JavaScript and iterates using tools (e.g., image generation, screenshot validation) until the website and tasks are functional. **WEBLICA-CACHE (bottom)**: We record browsing sessions capturing all HTTP traffic, then identify volatile parameters (e.g., timestamps, session tokens) that cause cache misses during playback. These are used to generate site-specific caching rules that strip volatile parameters from cache keys, enabling deterministic replay under complete network isolation.

2023) overlays. These overlay numbered bounding boxes on interactive elements to simplify action prediction. However, this introduces dependencies on accurate element detection and adds visual clutter that does not reflect natural web perception (Zheng et al., 2024). More recent work removes these aids entirely, building agents that operate on raw screenshots and predict actions as pixel coordinates (Qin et al., 2025; Wang et al., 2025a; Andreux et al., 2025; Awadallah et al., 2025; Gupta et al., 2026; OpenAI; Google). We follow this direction and train visual web agents with screenshot input and coordinate-based actions.

Data and Environments for Web Agents. Several efforts collect supervised fine-tuning (SFT) trajectories through human annotation or model-generated rollouts. Fara (Awadallah et al., 2025) develops a multi-agent data generation system that produces 145K trajectories across 70K domains. MolmoWeb (Gupta et al., 2026) combines over 100K synthetic task trajectories with 30K+ human demonstrations and GUI perception data. OpenCUA (Wang et al., 2025b) and AgentTrek (Xu et al., 2024a) similarly collect demonstration data for web tasks. While valuable, SFT data alone provides limited support for the exploration and trial-and-error learning that RL training enables.

Synthetic environments offer an alternative by enabling RL training in controlled settings. WebArena (Zhou et al., 2023; Koh et al., 2024) provide self-hosted websites that simulate e-commerce, forums, and content management systems. WebRL (Qi et al., 2024) and AgentGym-RL (Xi et al., 2025) build on these for RL training, yet they cover only a handful of domains and do not capture the diversity of the real web. Other work improves RL infrastructure and algorithms for such environments (Lu et al., 2025; Wei et al., 2025). Closer to our work, recent methods use LLMs to synthesize interactive websites (Zhang et al., 2026; Fan et al., 2026; Wu et al., 2026; Chae et al., 2026) or lightweight web replicas of mobile applications (Cao et al., 2026) with programmatic rewards, though at a limited training scale.

Recent work has explored scaling task generation. InstaV3 (Trabucco et al., 2025) develops an LLM-based pipeline to generate web navigation tasks across 146K live websites. WebGym (Bai et al., 2026) sources several datasets for RL training on live websites, but suffers from reproducibility issues and training instability due to timeouts and bot detection. Our framework addresses these limitations through LLM-based environment synthesis grounded in web navigation capabilities observed on real websites and diversified across website categories and visual styles, and further uses HTTP-level caching of real websites as a proxy for live-web training.

Evaluating Web Agents. Evaluation benchmarks for web agents span visual grounding and end-to-end task completion. For visual grounding, benchmarks like ScreenSpot-v2 (Wu et al., 2024), ScreenSpot-Pro (Li et al., 2025), and MMBench-GUI (Wang et al., 2025c) evaluate an agent's ability to localize and interact with UI elements. For task completion, benchmarks vary in their use of simulated versus real environments. World of Bits (Shi et al., 2017) was an early effort that cached HTTP traffic to create reproducible offline approximations of real websites, though at a small scale. WebArena (Zhou et al., 2023) and VisualWebArena (Koh et al., 2024) evaluate agents on self-hosted websites with programmatic success checking. While reproducible, they suffer from a sim-to-real gap. Benchmarks on real websites include GAIA (Mialon et al., 2023), WebVoyager (He et al., 2024), and Mind2Web (Deng et al., 2023), which test agents on live web tasks but face reproducibility challenges as websites change over time. WebVoyager additionally suffers from limited task diversity, with up to 51% of tasks solvable via search shortcuts. Online-Mind2Web (Xue et al., 2025) addresses these issues with a more realistic setup that evaluates agents on live websites using an LLM-as-Judge for task success. DeepShop (Lyu et al., 2025) and WebTailBench (Awadallah et al., 2025) further test agents on e-commerce and long-tail web tasks, respectively.

3 Framework

Figure 2 provides an overview of our framework. We first describe the agent formulation for visual web navigation (Section 3.1) and our LLM-based environment synthesis pipeline (Section 3.2). We then describe HTTP-level caching of real websites (Section 3.3).

3.1 Agent Formulation

We formulate web navigation as a partially observable Markov decision process (POMDP) defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, T, R)$, where $\mathcal{S}$ is the state space of the browser, $\mathcal{A}$ is the action space, $\mathcal{O}$ is the observation space, $T(s_{t+1} \mid s_t, a_t)$ is the transition function governing how the browser state changes in response to an action (implemented via Playwright), and R is the reward function (defined in Sec. 4). At each timestep t, the agent receives an observation $o_t \in \mathcal{O}$ and selects an action $a_t \in \mathcal{A}$ conditioned on the task instruction τ and history $o_{\leq t}$. See Fig. 3 for an example trajectory.

Observation Space. Each observation $o_t = (s_t, u_t)$ consists of a browser screenshot s_t rendered at 1280×720 pixels and the current URL u_t. Unlike approaches that rely on accessibility trees or DOM structures, our agents operate purely on visual input.

Action Space. We adopt a coordinate-based action space following recent work on visual web agents. Coordinate-based actions (`click`, `hover`) take pixel positions (x, y) as arguments, while other actions take task-specific arguments (text, keys, direction, etc.). The `stop` action terminates the episode and optionally returns a response. Please see Table 14 for the full action space.

Policy. The agent policy $\pi_\theta(a_t \mid o_{\leq t}, \tau)$ is parameterized by a vision-language model and follows a ReAct-style (Yao et al., 2022b) framework. At each step, the model produces a reasoning trace r_t analyzing the current observation, then selects an action a_t. Both reasoning traces and actions are appended to the history for subsequent steps. We use Qwen3-VL-Instruct as our base model, which supports coordinate-based action prediction without set-of-marks or other visual annotations.

3.2 WebLica-Synth: LLM-Based Environment Synthesis

Recent agentic coding tools such as Claude Code (Anthropic, 2025a) have demonstrated strong autonomous coding capabilities. Given task descriptions and verification criteria, these systems can

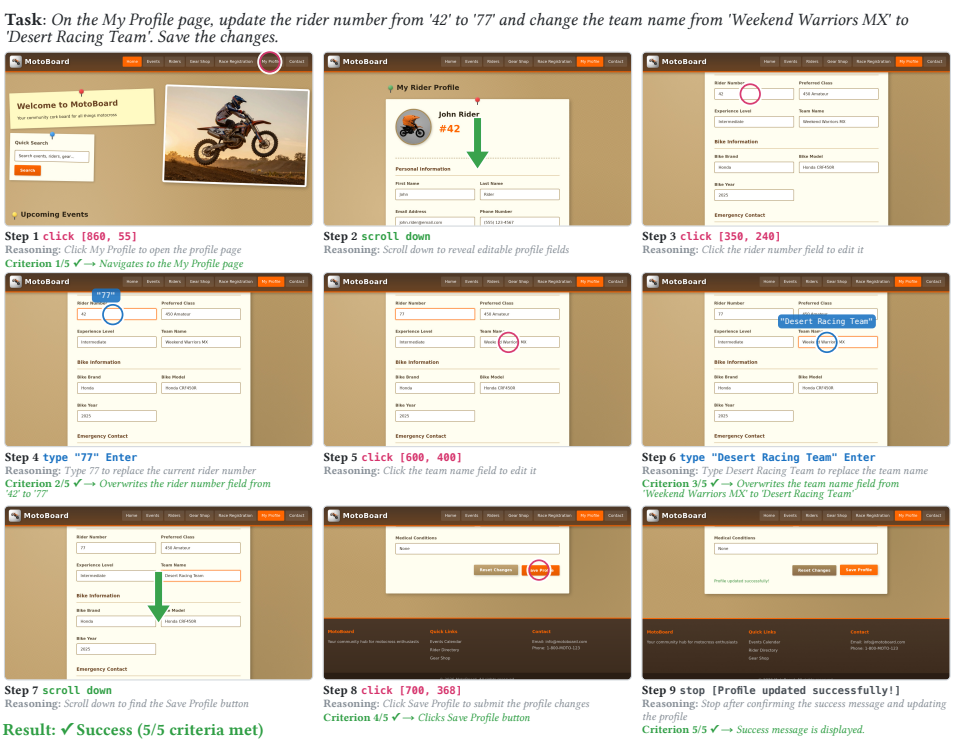


Figure 3: Example trajectory of WEBLICA-8B solving a data entry task in a WEBLICA-SYNTH environment, evaluated by an LLM judge against task-specific criteria (more examples in Appendix E).

work independently until success requirements are satisfied. We leverage such tools to generate synthetic web environments at scale. Because each environment is defined by its own code, it exposes its full interactive state, including stateful workflows such as carts and multi-step forms. Its tasks are written together with the environment rather than limited to what an existing agent can already solve.

Capability extraction. We target the set of broad web navigation capabilities used across common websites and reflected in benchmarks like Online-Mind2Web (Xue et al., 2025). To identify these capabilities, we first collect trajectories from Qwen3-VL-32B-Instruct attempting the Online-Mind2Web tasks. We then use GPT-5.2 (OpenAI, 2025a) to analyze the successful and failed trajectory screenshots and extract coarse web interaction capabilities apparent in each. This yields 19,721 fine-grained capabilities (*e.g., tab interface navigation, open dropdown menu, etc.*), which we aggregate into 144 higher-level capability groups (*e.g., navigation, form input, date selection, map interaction*). Note that we use these trajectories only to derive *generic interaction capabilities*. Synthesis does not use benchmark websites, tasks, or answers, and gains generalize to other live-web benchmarks that we do not use for capability extraction (Section 5.2). Please see Appendix B.2 for more details.

Diverse website generation. We use Claude Code (Opus 4.5 (Anthropic, 2025b)) to automatically generate self-contained web environments, writing static HTML, JavaScript, and CSS without any external framework dependencies. Each generation is parameterized by a target capability group, a website category randomly sampled from a pool of 1,160 domains (*e.g., aviation, banking, yoga studio, zoology, etc.*), and a visual style sampled from 961 options (*e.g., Editorial, Minimalist, Skeuomorphic, Duotone, etc.*). See Fig. 18 for an overview of the distribution of capabilities, domains, and visual styles. Sampling website categories and visual styles is important for diversity, as we found that without them, generated websites converge to a narrow visual style and content. For each website we generate at least 10 tasks of varying difficulty that require performing the target capability and other interactions present on the site, together with task-specific evaluation criteria following InstaV3 (Trabucco et al., 2025). The websites are static and have no backend, and we use JavaScript's `localStorage` feature to save values (e.g., items in cart) during a session. We use Z-Image-Turbo (Team et al., 2025) to automatically generate relevant visual assets for each website, such as product images, banners, etc. To ensure quality, we instruct Claude Code to self-validate each

site by taking screenshots via Playwright and iterating until the output satisfies the desired quality, i.e., that the website and tasks are functional and that there are no CSS issues. We describe the full generation workflow, including all prompts, in Appendix B, and statistics in Appendix G.

Environment server. Serving synthetically generated websites instead of training on the real web also enables much faster environment interactions, with environment setup/reset and action-to-screenshot times around 50 to 150ms. Hosting environments locally enables us to skip `sleep` instructions to wait for the network to stabilize, and we additionally use Playwright's animation skipping feature. Overall we observed a 30-40% end-to-end RL training speed improvement.

Environments and tasks. In total, we generated 310 sites covering the higher-level capability groups, and 2,500 sites targeting the fine-grained capabilities sorted by frequency. We call the resulting collection of *synthetic* web environments WEBLICA-SYNTH. From this full set, we reserve 2,560 web environments for training covering 44,227 tasks (WEBLICA-train), and 250 sites for validation covering 500 tasks (WEBLICA-val). Please see Fig. 1 as well as Figs. 8 to 10 for random examples of synthetically generated websites.

3.3 WEBLICA-CACHE: A PROXY FOR LIVE-WEB TRAINING

Why not train directly on the live web? As discussed in Section 1, live websites are slow, brittle, and irreproducible, and exploratory actions can have unintended side effects. A natural alternative is to record real websites once and replay them offline. We build such a proxy for live-web training at scale, which also lets us compare synthesized environments with real websites under an identical, reproducible setup. Please see Appendix C for details.

Recording and replay. We implement HTTP-level caching using Playwright to record and replay web interactions. During recording, we capture all HTTP traffic from multiple Qwen3-VL-32B-Instruct rollouts per task and index responses by normalized request signatures. The key challenge is handling volatile parameters (e.g., timestamps, session tokens) that change between visits and cause cache misses. We identify them automatically by fuzzy-matching cache misses during playback against the recording, and generate site-specific rules that strip them from cache keys. We then validate each session through playback under complete network isolation and keep only sessions in which the task remains completable.

Environments and tasks. We use InstaV3 (Trabucco et al., 2025), which provides web navigation tasks across 146K websites, as our task pool, yielding 15.6K verified cached environments, which we call WEBLICA-CACHE. By construction, a cached website only supports the interactions that were recorded and only contains tasks the recording agent could solve. Actions outside the recorded support are treated as website failures and excluded from training.

4 TRAINING

RL Environments. We subsample 10K tasks from each environment type. Our WEBLICA models are trained with RL on WEBLICA-SYNTH environments (Section 3.2). For comparison, we train the same model on WEBLICA-CACHE (Section 3.3) with an identical recipe.

Reward. As many web navigation tasks are open-ended and cannot be evaluated programmatically or via string matching, we implement an LLM-as-judge (Zheng et al., 2023) reward mechanism. Given a task description and criteria, the agent's action sequence, and resulting screenshots, we prompt GPT-4o (Hurst et al., 2024) to assess whether the agent successfully completed the task. This enables training on the full diversity of web tasks beyond those with programmatic verification. We validate the LLM judge by measuring agreement with human evaluations, finding 88% agreement (see Appendices F.4 and F.5 for details and further analysis).

SFT Warm-Start. We generate supervised fine-tuning data by collecting trajectories from a Qwen3-VL-32B-Instruct agent on InstaV3 queries. As not all rollouts succeed due to agent failures or website-specific issues such as changed task criteria or timeouts, we filter trajectories using the LLM judge to retain only successful completions. To increase coverage, we sample multiple rollouts per query with diverse sampling parameters, yielding 51.7K SFT trajectories in total (Figure 19).

Table 1: Comparison across web navigation benchmarks. Avg. is the mean over the three live benchmarks. Baseline results are reported from (Awadallah et al., 2025; Gupta et al., 2026). Italic Online-Mind2Web results are from the auto-eval leaderboard. With only 60 total steps compared to $\geq$100 for baselines, WEBLICA-8B already achieves the best average among non-proprietary models (above dashed line), and further improves with additional test-time compute (below dashed line). We **bold** the best non-proprietary result per column and underline the second best. † Fara-7B and MolmoWeb-8B allow up to 5 and 10 retries per trajectory.

		Live Web Navigation				*Synthetic*
Model	Total # Steps	Online-Mind2Web	DeepShop	WebTailBench	Avg.	WEBLICA-val
API-only						
OpenAI CUA	100	*58.3*	24.7	25.7	36.2	–
Gemini CUA	100	*57.3*	62.0	63.0	60.8	–
Yutori Navigator (Team, 2025)	–	*64.7*	–	–	–	–
Open-weight						
Qwen3-VL-8B-Instruct	30	28.6	24.1	21.8	24.8	56.9
Qwen3-VL-32B-Instruct	30	32.8	33.8	25.2	30.6	65.3
UI-TARS-1.5-7B (Qin et al., 2025)	100	31.3	11.6	19.5	20.8	–
GLM-4.1V-9B-Thinking (Hong et al., 2025)	100	33.9	32.0	22.4	29.4	–
Fara-7B (Awadallah et al., 2025)	$\geq$100†	34.1	26.2	38.4	32.9	–
MolmoWeb-8B (Gupta et al., 2026)	$\geq$100†	35.3	42.3	**49.5**	42.4	–
Ours (pass@k)						
WEBLICA-8B (k=1)	30	39.2	34.2	33.5	35.6	70.6
WEBLICA-8B (k=2)	**60**	**50.3**	**45.4**	47.0	**47.6**	**79.0**
WEBLICA-8B (k=4)	120	60.5	55.9	60.3	58.9	84.7
WEBLICA-8B (k=8)	240	68.8	65.8	72.2	68.9	88.6

Training on this data gives an initialization with reasonable navigation behavior, which we compare against RL directly from the base model in Section 5.4.

RL Training. We train using Dr. GRPO (Liu et al., 2025), a variant of GRPO (Shao et al., 2024), with the LLM-as-Judge reward. During RL, we oversample rollouts and filter to retain groups with mixed success signals, similar to DAPO (Yu et al., 2025), and apply fallback mechanisms to handle errors or timeouts, ensuring stable training. Please see Appendix F for details.

5 EXPERIMENTS

We evaluate WEBLICA on web navigation and UI grounding benchmarks. After describing our setup (Sec. 5.1), we present main results against existing agents (Sec. 5.2), analyze test-time scaling (Sec. 5.3), and ablate training stages, environment choices, and the effect on grounding (Sec. 5.4).

5.1 SETUP

Evaluation. We evaluate on four web navigation benchmarks. Online-Mind2Web (Xue et al., 2025) tests agents on diverse live websites. DeepShop (Lyu et al., 2025) focuses on e-commerce tasks. WebTailBench (Awadallah et al., 2025) tests agents on long-tail web tasks. We use the official judges provided by each benchmark for evaluation. For WebTailBench, we follow MolmoWeb (Gupta et al., 2026) and use the WebVoyager (He et al., 2024) judge. We also evaluate on WEBLICA-val, 500 held-out tasks on 250 unseen synthesized environments, to measure in-distribution generalization. We found WEBLICA-val has strong correlation with performance on live-web tasks (Appendix D.4). We study test-time scaling by running $k \in \{1, 2, 4, 8\}$ independent attempts and reporting pass@k, where total step budget scales linearly with k. Unlike retry-based evaluation in prior work (Awadallah et al., 2025; Gupta et al., 2026), where failures attributed to the environment are selectively discarded, this provides a more reliable estimate of task success and a transparent measure of total test-time compute. We repeat each experiment three times and report mean accuracy; standard deviations are reported in plots and in Appendix D. We additionally evaluate UI grounding on MMBench-GUI (Wang et al., 2025c), ScreenSpot-v2 (Wu et al., 2024), and ScreenSpot-Pro (Li et al., 2025) (Sec. 5.4).

Baselines. We compare against open-weight models including UI-TARS-1.5-7B (Qin et al., 2025), GLM-4.1V-9B-Thinking (Hong et al., 2025), Fara-7B (Awadallah et al., 2025), and MolmoWeb-8B (Gupta et al., 2026), as well as API-based agents including OpenAI computer-use-preview,

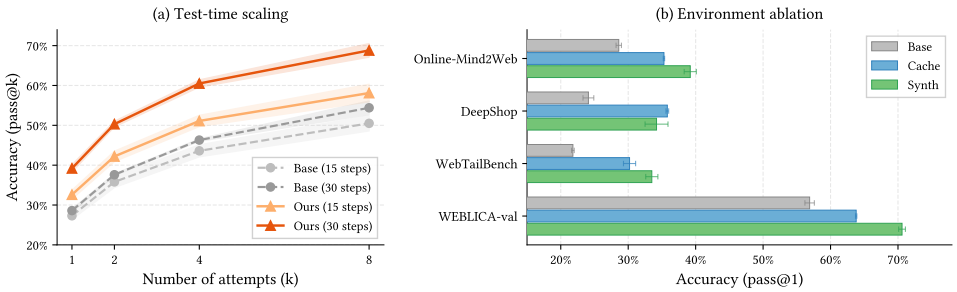


Figure 4: **(a)** Test-time scaling on Online-Mind2Web. WEBLICA-8B benefits from both increasing the action budget (steps per episode) and parallel attempts (pass@k). The base Qwen3-VL-8B, in contrast, shows minimal gains from longer episodes, suggesting that RL training enables the model to effectively leverage additional steps. **(b)** Environment ablation across four benchmarks. Both training configurations improve substantially over the base model, with synthesized environments outperforming cache on most benchmarks.

Gemini computer-use-preview, and Yutori Navigator (Team, 2025). We also evaluate the base Qwen3-VL-8B-Instruct and Qwen3-VL-32B-Instruct as reference.

5.2 MAIN RESULTS

Table 1 presents results across four benchmarks and Figure 3 shows a qualitative example (more in Appendix E). We summarize the key observations below.

Training on WEBLICA data improves the base model by 44%. At pass@1 with 30 steps, WEBLICA-8B improves the base Qwen3-VL-8B from 24.8% to 35.6% average across the three live benchmarks, an absolute gain of 10.8 percentage points. On WEBLICA-val, our held-out evaluation set of synthesized environments, training improves the base from 56.9% to 70.6%. We observe consistent improvements at 2B and 4B model sizes as well (Figure 1 and Appendix D.1).

WEBLICA-8B outperforms open-weight baselines with less test-time compute. With only 60 total steps, WEBLICA-8B surpasses the best open-weight baseline average (47.6% vs 42.4% for MolmoWeb-8B at $\geq$100 steps), and scales to 58.9% and 68.9% with 120 and 240 steps respectively.

WEBLICA-8B is competitive with API models. At pass@1 with only 30 steps, WEBLICA-8B nearly matches OpenAI computer-use-preview on average (35.6% vs 36.2% at 100 steps). Additional test-time compute leads to further improvements: with 120 total steps, WEBLICA-8B reaches 58.9%, approaching Gemini computer-use-preview (60.8%), and with 240 steps reaches 68.9%. These results demonstrate the effectiveness of training on synthetic WEBLICA environments, producing strong visual web agents without relying on large-scale human demonstrations.

5.3 TEST-TIME SCALING

We study test-time scaling along two axes: the number of parallel attempts (pass@k) and the per-episode action budget. Figure 4a compares WEBLICA-8B and the base Qwen3-VL-8B-Instruct on Online-Mind2Web. WEBLICA-8B improves consistently along both axes: increasing the action budget from 15 to 30 steps improves pass@1 from 32.6% to 39.2%, and pass@8 at 30 steps per attempt reaches 68.8%. In contrast, the base model shows minimal gains from a larger action budget, with its 15- and 30-step curves nearly overlapping. This gap suggests that RL training is key to unlocking effective use of longer episodes.

5.4 ANALYSIS

Environment Type. We compare RL on WEBLICA-SYNTH and on WEBLICA-CACHE under the same recipe (Figure 4b). Training on cached real websites already yields large gains: it improves

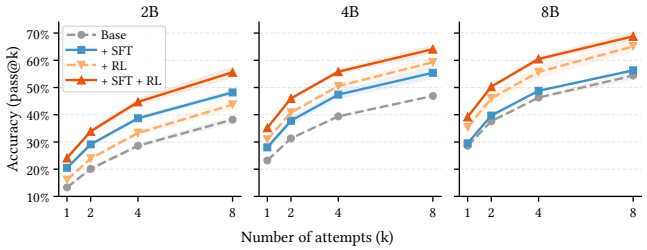


Figure 5: Training stage ablation on Online-Mind2Web across three model sizes. SFT and RL each improve over the base model, and their combination performs best at all scales.

Table 2: WEBLICA-8B preserves grounding. SS: ScreenSpot; MMB: MMBench-GUI.

Model	MMB	SS-v2	SS-Pro
Fara-7B	—	89.3	—
MolmoWeb-Ground-8B	—	91.8	—
Qwen3-VL-8B	82.85	93.95	54.71
WEBLICA-8B	**83.74**	**94.50**	**55.28**

over the base model on all four benchmarks and even surpasses the 32B teacher that recorded the sessions on all three live benchmarks (Appendix D.2). Synthesized environments nevertheless perform better overall (35.6% vs. 33.8% live average), with the largest gains on Online-Mind2Web (39.2% vs. 35.3%) and WebTailBench (33.5% vs. 30.2%), while WEBLICA-CACHE is slightly better on DeepShop (35.8% vs. 34.2%). This likely reflects the limits of caching: cached websites only support recorded interactions and tasks the recording agent could solve (Section 3.3). Our initial attempts with different mixture ratios of cached and synthesized environments did not yield further improvements; we expect that designing effective training curricula across environment types could lead to additional gains.

Training Stages and Model Size. Figure 5 compares the contribution of each training stage across model sizes on Online-Mind2Web. Both SFT and RL provide consistent gains at all scales. At 2B, the full SFT+RL pipeline improves pass@1 from 13.3% to 24.1%, at 4B from 23.2% to 35.2%, and at 8B from 28.6% to 39.2%. RL on top of SFT further improves over SFT alone at every scale. Comparing SFT→RL with Base→RL shows that SFT initialization is critical for smaller models, but SFT data has diminishing returns at larger scales.

Effect on Grounding. As visual grounding is a core capability for web agents, we evaluate whether our training pipeline affects it, given that our training data contains no grounding-specific data. Table 2 shows that grounding is preserved after training, with modest improvements across all three benchmarks. Since grounding performance remains comparable, the gains on web navigation benchmarks stem from improved navigation behavior rather than better visual grounding.

6 CONCLUSION AND LIMITATIONS

We presented WEBLICA, a framework for building scalable and reproducible training environments for visual web agents. At its core, WEBLICA synthesizes interactive websites with coding agents, grounded in real-world websites and core web navigation skills. We further build WEBLICA-CACHE, which caches real websites at the HTTP level to train on them without live interaction. Training on either environment type substantially improves over the base model, with purely synthetic RL performing best overall. Our best model, WEBLICA-8B, outperforms open-weight baselines of similar size with fewer inference steps and scales favorably with additional test-time compute. We outline some limitations of our work and future directions.

- **Single-turn tasks.** Our setup evaluates agents on isolated tasks with a fixed goal defined at the start of the episode. Real-world web usage involves multi-turn sessions with evolving goals and human-in-the-loop interaction. Extending to these settings is an exciting direction.
- **Training.** We obtain promising gains using a vanilla RL training formulation. Further exploration of the RL framework, including long-horizon RL and scaling RL compute (Khatri et al., 2025), and richer SFT data such as error recovery trajectories are interesting directions for improving web agent training.
- **Beyond web.** Our framework currently targets web navigation. Extending it to other GUI environments such as mobile and desktop applications to build generalist computer-use agents is a promising direction.

AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic data sets, support data analysis, and clean datasets, as components of our method: Claude Code with Claude Opus 4.5 generates the WEBLICA-SYNTH environments and their tasks, Claude Opus 4.5 generated the pools of website categories and visual styles, Z-Image-Turbo generates the images of synthesized websites, Qwen3-VL-32B-Instruct generates the SFT trajectories and records the WEBLICA-CACHE sessions, GPT-5.2 extracts, normalizes, and groups web navigation capabilities from agent trajectories, and GPT-4o judges task success to provide RL rewards and to filter SFT trajectories and cached sessions (Sections 3.2, 3.3 and 4; Appendix B). We have not used generative AI tools to propose or refine hypotheses, design experiments, or implement methods, and developing theoretical models, formulating or proving mathematical claims, and translation are not applicable to this work. Additionally, we used generative AI tools to draft and edit text to improve readability. We have reviewed all AI-assisted work: the authors checked all interpretations, verified all reported numbers against our experimental records, and checked all references against the cited papers. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Appendix B provides full details of the WEBLICA-SYNTH pipeline, including capability extraction, seed pools, the coding-agent harness and tools, task and criteria generation, all prompts, per-site statistics, and generation costs. The synthesized environments are static HTML, CSS, and JavaScript without a backend, so training on them requires no proprietary model, and generation is a one-time step that other capable coding agents could also perform. Appendix C details the construction of WEBLICA-CACHE. Section 4 and Appendix F describe the training recipe, hyperparameters, compute, action space, and judge prompt, and Appendix F.5 shows that open-weight judges perform on par with GPT-4o. Our pipeline builds on open-weight models (Qwen3-VL (Yang et al., 2025a)), open-source tools (Playwright and veRL (Sheng et al., 2024)), and a public task dataset (InstaV3 (Trabucco et al., 2025)). We evaluate on public benchmarks with their official judges, repeat each evaluation three times, and report standard deviations (Section 5.1; Appendix D).

REFERENCES

Mathieu Andreux, Breno Baldas Skuk, Hamza Benchekroun, Emilien Biré, Antoine Bonnet, Riaz Bordie, Nathan Bout, Matthias Brunel, Pierre-Louis Cedoz, Antoine Chassang, et al. Surfer-h meets holo1: Cost-efficient web agent powered by open weights. *arXiv preprint arXiv:2506.02865*, 2025.

Anthropic. Claude computer use. `https://www.anthropic.com/news/3-5-models-and-computer-use`, 2024. Accessed: 2026-04-30.

Anthropic. Claude code. `https://www.anthropic.com/claude-code`, 2025a. Accessed: 2026-04-30.

Anthropic. Introducing Claude Opus 4.5. `https://www.anthropic.com/news/claude-opus-4-5`, November 2025b. Accessed: 2026-04-30.

Ahmed Awadallah, Yash Lara, Raghav Magazine, Hussein Mozannar, Akshay Nambi, Yash Pandya, Aravind Rajeswaran, Corby Rosset, Alexey Taymanov, Vibhav Vineet, et al. Fara-7b: An efficient agentic model for computer use. *arXiv preprint arXiv:2511.19663*, 2025.

Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. *Advances in Neural Information Processing Systems*, 37:12461–12495, 2024.

Hao Bai, Alexey Taymanov, Tong Zhang, Aviral Kumar, and Spencer Whitehead. Webgym: Scaling training environments for visual web agents with realistic tasks. *arXiv preprint arXiv:2601.02439*, 2026.

Yuan Cao, Dezhi Ran, Mengzhou Wu, Yuzhe Guo, Xin Chen, Ang Li, Gang Cao, Gong Zhi, Hao Yu, Linyi Li, et al. Gui-genesis: Automated synthesis of efficient environments with verifiable rewards for gui agent post-training. *arXiv preprint arXiv:2602.14093*, 2026.

Hyungjoo Chae, Jungsoo Park, and Alan Ritter. Safe and scalable web agent learning via recreated websites. *arXiv preprint arXiv:2603.10505*, 2026.

Khaoula Chehbouni, Mohammed Haddou, Jackie CK Cheung, and Golnoosh Farnadi. Neither valid nor reliable? Investigating the use of LLMs as judges. *Advances in Neural Information Processing Systems*, 38, 2026.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9313–9332, 2024.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Sicheng Fan, Qingyun Shi, Shengze Xu, Shengbo Cai, Tieyong Zeng, Li Ling, Yanyi Shang, and Dehan Kong. Webfactory: Automated compression of foundational language intelligence into grounded web agents. *arXiv preprint arXiv:2603.05044*, 2026.

Div Garg, Diego Caples, Andis Draguns, Nikil Ravi, Pranav Putta, Naman Garg, Prannay Hebbar, Youngchul Joo, Jindong Gu, Charles London, et al. REAL: Benchmarking autonomous agents on deterministic simulations of real websites. *Advances in Neural Information Processing Systems*, 38, 2026.

Google. Gemini computer use. `https://ai.google.dev/gemini-api/docs/computer-use`. Accessed: 2026-04-30.

Boyu Gou, Demi 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. In *International Conference on Learning Representations*, volume 2025, pp. 30851–30883, 2025.

Tanmay Gupta, Piper Wolters, Zixian Ma, Peter Sushko, Rock Yuren Pang, Diego Llanes, Yue Yang, Taira Anderson, Boyuan Zheng, Zhongzheng Ren, et al. Molmoweb: Open visual web agent and open data for the open web. *arXiv preprint arXiv:2604.08516*, 2026.

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

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

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

Wenyi Hong, Wenmeng Yu, Xiaotao Gu, Guo Wang, Guobing Gan, Haomiao Tang, Jiale Cheng, Ji Qi, Junhui Ji, Lihang Pan, et al. Glm-4.1 v-thinking: Towards versatile multimodal reasoning with scalable reinforcement learning. *arXiv e-prints*, pp. arXiv–2507, 2025.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, pp. 1–3, 2025.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Devvrit Khatri, Lovish Madaan, Rishabh Tiwari, Rachit Bansal, Sai Surya Duvvuri, Manzil Zaheer, Inderjit S Dhillon, David Brandfonbrener, and Rishabh Agarwal. The art of scaling reinforcement learning compute for llms. *arXiv preprint arXiv:2510.13786*, 2025.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905, 2024.

Kaixin Li, Ziyang Meng, Hongzhan Lin, Ziyang Luo, Yuchen Tian, Jing Ma, Zhiyong Huang, and Tat-Seng Chua. Screenspot-pro: Gui grounding for professional high-resolution computer use. In *Proceedings of the 33rd ACM International Conference on Multimedia*, pp. 8778–8786, 2025.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, Karolina Stańczak, Peter Shaw, Christopher J Pal, and Siva Reddy. Agentrewardbench: Evaluating automatic evaluations of web agent trajectories. *arXiv preprint arXiv:2504.08942*, 2025.

Yuxuan Lu, Ziyi Wang, Jing Huang, Hui Liu, Jiri Gesi, Yan Han, Shihan Fu, Tianqi Zheng, Xianfeng Tang, Chen Luo, et al. Webserv: A full-stack and rl-ready web environment for training web agents at scale. *arXiv preprint arXiv:2510.16252*, 2025.

Yougang Lyu, Xiaoyu Zhang, Lingyong Yan, Maarten de Rijke, Zhaochun Ren, and Xiuying Chen. Deepshop: A benchmark for deep research shopping agents. *arXiv preprint arXiv:2506.02839*, 2025.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Shikhar Murty, Hao Zhu, Dzmitry Bahdanau, and Christopher D Manning. Nnetnav: Unsupervised learning of browser agents through environment interaction in the wild. *arXiv preprint arXiv:2410.02907*, 2024.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. Computer use. `https://developers.openai.com/api/docs/guides/tools-computer-use`. Accessed: 2026-04-30.

OpenAI. Introducing GPT-5.2. `https://openai.com/index/introducing-gpt-5-2/`, December 2025a. Accessed: 2026-04-30.

OpenAI. Codex. `https://chatgpt.com/codex/`, 2025b. Accessed: 2026-04-30.

Tianyue Ou, Frank F Xu, Aman Madaan, Jiarui Liu, Robert Lo, Abishek Sridhar, Sudipta Sengupta, Dan Roth, Graham Neubig, and Shuyan Zhou. Synatra: Turning indirect knowledge into direct demonstrations for digital agents at scale. *Advances in Neural Information Processing Systems*, 37: 91618–91652, 2024.

Vardaan Pahuja, Yadong Lu, Corby Rosset, Boyu Gou, Arindam Mitra, Spencer Whitehead, Yu Su, and Ahmed Hassan. Explorer: Scaling exploration-driven web trajectory synthesis for multimodal web agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 6300–6323, 2025.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Wenyi Zhao, Yu Yang, Xinyue Yang, Jiadai Sun, Shuntian Yao, et al. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. *arXiv preprint arXiv:2411.02337*, 2024.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, volume 2025, pp. 406–441, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In *International Conference on Machine Learning*, pp. 3135–3144. PMLR, 2017.

Sanjari Srivastava, Gang Li, Cheng Chang, Rishu Garg, Manpreet Kaur, Charlene Lee, Yuezhang Li, Yining Mao, Juan Ignacio Cases Martin, Yanan Xie, et al. Warc-bench: Web archive based benchmark for gui subtask executions. In *International Conference on Learning Representations*, volume 2026, pp. 26166–26191, 2026.

Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, et al. Os-genesis: Automating gui agent trajectory construction via reverse task synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5555–5579, 2025.

The Yutori Team. Introducing navigator. `https://yutori.com/blog/introducing-navigator`, 2025. Yutori Blog.

Z-Image Team, Huanqia Cai, Sihan Cao, Ruoyi Du, Peng Gao, Steven Hoi, Zhaohui Hou, Shijie Huang, Dengyang Jiang, Xin Jin, Liangchen Li, Zhen Li, Zhong-Yu Li, David Liu, Dongyang Liu, Junhan Shi, Qilong Wu, Fengyi Yu, Chi Zhang, Shifeng Zhang, and Shilin Zhou. Z-image: An efficient image generation foundation model with single-stream diffusion transformer. *ArXiv*, abs/2511.22699, 2025. URL `https://api.semanticscholar.org/CorpusID:283438115`.

Brandon Trabucco, Gunnar Sigurdsson, Robinson Piramuthu, and Ruslan Salakhutdinov. Insta: Towards internet-scale training for agents. *arXiv preprint arXiv:2502.06776*, 2025.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, et al. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, et al. Opencua: Open foundations for computer-use agents. *arXiv preprint arXiv:2508.09123*, 2025b.

Xuehui Wang, Zhenyu Wu, JingJing Xie, Zichen Ding, Bowen Yang, Zehao Li, Zhaoyang Liu, Qingyun Li, Xuan Dong, Zhe Chen, et al. Mmbench-gui: Hierarchical multi-platform evaluation framework for gui agents. *arXiv preprint arXiv:2507.19478*, 2025c.

Zhepei Wei, Wenlin Yao, Yao Liu, Weizhi Zhang, Qin Lu, Liang Qiu, Changlong Yu, Puyang Xu, Chao Zhang, Bing Yin, et al. WebAgent-R1: Training web agents via end-to-end multi-turn reinforcement learning. *arXiv preprint arXiv:2505.16421*, 2025.

Yifan Wu, Yiran Peng, Yiyu Chen, Jianhao Ruan, Zijie Zhuang, Cheng Yang, Jiayi Zhang, Man Chen, Yenchi Tseng, Zhaoyang Yu, et al. Autowebworld: Synthesizing infinite verifiable web environments via finite state machines. *arXiv preprint arXiv:2602.14296*, 2026.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, et al. Os-atlas: A foundation action model for generalist gui agents. *arXiv preprint arXiv:2410.23218*, 2024.

Zhiheng Xi, Jixuan Huang, Chenyang Liao, Baodai Huang, Honglin Guo, Jiaqi Liu, Rui Zheng, Junjie Ye, Jiazheng Zhang, Wenxiang Chen, et al. Agentgym-rl: Training llm agents for long-horizon decision making through multi-turn reinforcement learning. *arXiv preprint arXiv:2509.08755*, 2025.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Tianbao Xie, Jiaqi Deng, Xiaochuan Li, Junlin Yang, Haoyuan Wu, Jixuan Chen, Wenjing Hu, Xinyuan Wang, Yuhui Xu, Zekun Wang, et al. Scaling computer-use grounding via user interface decomposition and synthesis. *arXiv preprint arXiv:2505.13227*, 2025.

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. *arXiv preprint arXiv:2412.09605*, 2024a.

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous gui interaction. *arXiv preprint arXiv:2412.04454*, 2024b.

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. *arXiv preprint arXiv:2504.01382*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v. *arXiv preprint arXiv:2310.11441*, 2023.

Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, et al. Gta1: Gui test-time scaling agent. *arXiv preprint arXiv:2507.05791*, 2025b.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022b.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Yutori. The bitter lesson for web agents. `https://yutori.com/blog/the-bitter-lesson-for-web-agents`, December 2025. Accessed: 2026-04-30.

Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. Infiniteweb: Scalable web environment synthesis for gui agent training. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 28465–28492, 2026.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. Gpt-4v (ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*, 2024.

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

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

Yifei Zhou, Qianlan Yang, Kaixiang Lin, Min Bai, Xiong Zhou, Yu-Xiong Wang, Sergey Levine, and Li Erran Li. Proposer-agent-evaluator (pae): Autonomous skill discovery for foundation model internet agents. In *Forty-second International Conference on Machine Learning*, 2025.

APPENDIX

A Extended Related Work **17**

B WEBLICA-SYNTH: Generation Pipeline and Prompts **18**

B.1 Overview . . . 18
B.2 Capability extraction . . . 18
B.3 Seed sampling . . . 19
B.4 Website generation and self-validation . . . 20
B.5 Task and criteria generation . . . 21
B.6 Prompts . . . 22
B.7 Seed pools . . . 31
B.8 Environment generation cost . . . 33

C WEBLICA-CACHE: Construction Details **33**

C.1 Recording and replay . . . 33
C.2 Automated rule generation . . . 33
C.3 Construction funnel . . . 33
C.4 Out-of-support actions . . . 34
C.5 Training on WEBLICA-CACHE . . . 34
C.6 Limitations . . . 34

D Additional Quantitative Results **34**

D.1 Results across model sizes . . . 34
D.2 Comparison with the SFT teacher and WEBLICA-CACHE . . . 34
D.3 Downstream performance during training . . . 35
D.4 WEBLICA-val and live-web performance . . . 37

E Additional Qualitative Results **38**

E.1 WEBLICA-SYNTH visualizations . . . 38
E.2 Solved trajectory examples . . . 41

F Additional Training Details **45**

F.1 Training hyperparameters . . . 45
F.2 Compute . . . 46
F.3 Action space . . . 46
F.4 Training reward judge prompt . . . 47
F.5 Judge validation . . . 47

G Data Statistics **48**

A Extended Related Work

Visual Web and GUI Agents. Web agents have evolved from text-only models operating on HTML, DOM, or accessibility trees (Yao et al., 2022a; Deng et al., 2023; Zhou et al., 2023; Gur et al., 2023) to vision-language models that perceive rendered pages (Koh et al., 2024; He et al., 2024; Hong et al., 2024). Early visual agents relied on set-of-marks prompting (Yang et al., 2023; Zheng et al., 2024), whereas recent agents act directly on raw screenshots by predicting pixel coordinates (Qin et al., 2025; Wang et al., 2025a; Andreux et al., 2025; Awadallah et al., 2025; Gupta et al., 2026; Anthropic, 2024; OpenAI; Google). A related line of work improves GUI grounding, i.e., mapping instructions to screen locations, through large-scale grounding data and dedicated models (Wu et al., 2024; Xie et al., 2025; Cheng et al., 2024; Gou et al., 2025; Xu et al., 2024b), and through test-time scaling of action selection (Yang et al., 2025b). We follow the screenshot-only, coordinate-based setting and show that training on WEBLICA environments preserves grounding performance (Section 5.4).

Demonstration Data for Web Agents. A common way to train web agents is supervised fine-tuning on demonstrations. Fara (Awadallah et al., 2025) develops a multi-agent data generation system that produces 145K trajectories across 70K domains. MolmoWeb (Gupta et al., 2026) combines over 100K synthetic task trajectories with 30K+ human demonstrations and GUI perception data. OpenCUA (Wang et al., 2025b) and AgentTrek (Xu et al., 2024a) collect demonstrations from human annotators and web tutorials, respectively. Other work synthesizes trajectories through exploration or reverse task synthesis (Sun et al., 2025; Murty et al., 2024; Pahuja et al., 2025), or converts tutorials into demonstrations (Ou et al., 2024). InstaV3 (Trabucco et al., 2025) scales task generation to 146K live websites with an LLM-based pipeline, which we use as the task pool for our proxy for live-web training. While such data provides a reasonable initialization, it offers limited support for the exploration and trial-and-error learning that RL enables. We use such demonstrations to warm-start our models, and RL in our environments provides significantly further gains, especially at larger sizes (Section 5.4).

Environments for Reinforcement Learning. Self-hosted benchmarks such as WebArena (Zhou et al., 2023) and VisualWebArena (Koh et al., 2024) provide reproducible websites with programmatic success checks and have been used for RL training (Qi et al., 2024; Xi et al., 2025), but they cover only a handful of manually built domains. WebServ (Lu et al., 2025) makes such environments cheaper to serve at scale with lightweight containers, and WebAgent-R1 (Wei et al., 2025) proposes end-to-end multi-turn RL on them. Both are orthogonal to our focus on the diversity of the environments themselves. Similar environments exist for desktop and mobile GUIs (Xie et al., 2024; Rawles et al., 2025), and large-scale RL systems for GUI agents often rely on dedicated sandbox infrastructure (Wang et al., 2025a). An alternative is to train directly on live websites or devices with LLM-based rewards (Bai et al., 2026; 2024; Zhou et al., 2025); PAE (Zhou et al., 2025) additionally proposes tasks autonomously for screenshot-based web agents. This offers realism but suffers from timeouts, bot detection, changing content, and irreproducibility, and exploratory actions can have side effects on real websites. Thus, as an alternative, replaying recorded websites is considered to restore reproducibility. World of Bits (Shi et al., 2017) cached real websites for RL, WARC-Bench (Srivastava et al., 2026) builds GUI subtasks with programmatic rewards on web archives, and REAL (Garg et al., 2026) provides deterministic simulations of 11 real websites. WEBLICA-CACHE instead automates HTTP-level caching across thousands of websites in a scalable way (Appendix C).

Synthesizing Training Environments. Recent methods use LLMs to synthesize interactive environments for GUI agents, mostly with programmatic rewards or verifiers. InfiniteWeb (Zhang et al., 2026) generates websites from reference designs, GUI-Genesis (Cao et al., 2026) rebuilds mobile mini-app tasks as lightweight web applications, and WebFactory (Fan et al., 2026), AutoWebWorld (Wu et al., 2026), and VeriEnv (Chae et al., 2026) generate websites with verifiable tasks or trajectories. WEBLICA differs in three ways: (i) generation is grounded in web navigation capabilities observed on real websites and diversified across website categories and visual styles; (ii) it scales to thousands of environments and tasks; and (iii) it validates transfer on several public live-web benchmarks across model sizes. Furthermore, instead of programmatic checks, we reward agents with an LLM judge validated against human labels (Appendix F.5), which also covers open-ended tasks that such checks cannot verify.

Evaluation and LLM Judges. Beyond the benchmarks discussed in Section 2, evaluating open-ended tasks on live websites typically requires an LLM judge, as in Online-Mind2Web (Xue et al., 2025). LLM judges are also increasingly used as rewards for training web and GUI agents (Bai et al., 2026; 2024; Zhou et al., 2025), building on the broader LLM-as-a-judge paradigm (Zheng et al., 2023). However, their reliability varies across settings (Chehbouni et al., 2026). In particular, for web domain, AgentRewardBench (Lù et al., 2025) evaluates automatic judges of web-agent trajectories against expert labels. In our work, we validate our training judge against human labels on both training tasks and real-website trajectories (Appendix F.5) to ensure high quality training signal. Furthermore, we evaluate all live benchmarks with their benchmark-specific official judges rather than our training judge, and demonstrate generalization.

B WEBLICA-SYNTH: GENERATION PIPELINE AND PROMPTS

B.1 OVERVIEW

Each WEBLICA-SYNTH environment is the output of one autonomous Claude Code (Anthropic, 2025a) session running Claude Opus 4.5 (Anthropic, 2025b). Generation has four steps:

1. **Seed sampling.** We sample a target web interaction capability with a website category and, optionally, a visual style and a layout (Appendix B.3).
2. **Generation.** The agent receives a prompt built from the seed (Appendix B.6). It creates the website HTML, JavaScript, and CSS, generates image assets in parallel, and writes at least 10 tasks (Appendices B.4 and B.5).
3. **Self-validation.** The agent runs a verification script, inspects screenshots of its pages, and has a subagent review all tasks. It fixes the problems it finds and repeats until the site is complete.
4. **External check.** We rerun the verification script. If it fails, we restart the session with the error report, up to 10 times.

We generated 2,810 sites this way: 310 sites targeting 144 broad web capability groups, and 2,500 sites, each targeting one fine-grained capability. Table 3 summarizes per-site statistics.

Table 3: Per-site generation statistics of WEBLICA-SYNTH.

	Mean	Median
Tasks	17.3	17
HTML pages	6.0	6
Image files	68.8	59
Session time (min)	24.0	23
Agent turns	77.9	76

B.2 CAPABILITY EXTRACTION

We derive the target capabilities from agent trajectories on Online-Mind2Web (Xue et al., 2025) in three steps: extraction, normalization, and grouping. We use GPT-5.2 (OpenAI, 2025a) for all three. Appendix B.6 shows the prompts.

Trajectories. We ran Qwen3-VL-32B-Instruct on all 300 Online-Mind2Web tasks, with at least 8 rollouts per task, and kept all 15,495 trajectories. We include failed trajectories because they show capabilities the agent lacks.

Extraction. For each trajectory, GPT-5.2 sees up to 15 screenshots (the first, the last, and evenly spaced ones in between), the task, the visited URLs, and a condensed action sequence with excerpts of the agent's reasoning. It lists every interaction capability the trajectory demonstrates, with a snake_case name, a description, the UI element type, and whether it succeeded. The prompt defines a capability as an interaction pattern rather than a single action, e.g., using a date picker rather than

clicking a button. This yields 141,933 capability mentions (9.2 per trajectory) under 43,309 distinct names.

Normalization. Many names describe the same pattern, e.g., `navigate_dropdown_menu` and `use_dropdown_navigation`. The LLM clusters synonymous names and picks one general, verb-oriented name per cluster. It processes batches of 2,000 names together with their frequencies. A second pass over the resulting names merges duplicates across batches. This reduces 43,309 names to 19,721 fine-grained capabilities.

Grouping. The LLM then maps each fine-grained capability to a functional group, which yields a final 144 broad groups of capabilities. Table 4 shows examples, and Appendix B.7 lists all groups.

Table 4: Example capability groups, with their number of fine-grained capabilities and their most frequent members.

Group	Size	Most frequent fine-grained capabilities
`navigation`	2,942	`navigate_top_nav_menu`, `navigate_to_search_results_page`, `use_sidebar_navigation`
`content_filtering`	1,752	`use_faceted_filter_sidebar`, `select_checkbox_filter_option`, `open_filter_panel`
`form_input`	692	`fill_text_input_field`, `focus_text_input_field`, `fill_email_input_field`
`date_selection`	386	`open_date_picker`, `navigate_calendar_months`, `select_date_in_calendar`
`map_interaction`	297	`interact_with_map_widget`, `use_search_results_map_view`, `view_map_and_list_split_layout`
`captcha_handling`	265	`handle_captcha_challenge`, `use_captcha_language_selector`, `click_checkbox_captcha`

B.3 SEED SAMPLING

When asking a coding agent to generate a website for a given capability, we noticed that calling it repeatedly generated highly similar websites in terms of their domain and visual style. For that reason, we decided to seed the generation by randomly sampling a website domain and a visual style from a collection we assembled beforehand. A seed has up to four components (Table 5). Appendix B.7 lists the capability groups and samples from the other pools, and Fig. 18 shows the resulting distribution.

Capability. We built websites in two phases, first building a set that targets broad group-level capabilities, then extending it to a larger set of more fine-grained capabilities. Group-level sites cover each of the 144 capability groups two to four times. Fine-grained sites cover the 2,500 top-ranked of the 19,721 fine-grained capabilities, one site each. We rank a fine-grained capability by the number of raw capability names merged into it during normalization (Appendix B.2) and break ties randomly. The top 2,500 account for 76.3% of all capability occurrences in the analyzed trajectories.

Website category. We sample uniformly from 1,160 categories. Claude Opus 4.5 generated both this pool and the style pool below.

Visual style. With probability 0.5, the seed has no style and the prompt grants the agent full creative freedom. Otherwise, we sample uniformly from 961 styles. Each style consists of a name and a one-line description, e.g., *Art Deco: Geometric patterns, gold/brass accents, strong symmetry, ornamental sunburst details, 1920s luxury*. The prompt presents the style as a high-level inspiration only.

Layout. Fine-grained sites also receive a layout with probability 0.5. We compose it from 11 dimensions (Table 6). Each of the 4 required dimensions contributes one uniformly sampled option, and each of the 7 optional dimensions does so with probability 0.6. An example layout is *Floating Center-Bottom / Split Master-Detail / Paginated Listing With Page Controls*, described as *navigation position: floating center-bottom; navigation style: ribbon toolbar with grouped actions; primary content layout: split master-detail (list + detail); page model: paginated listing with page controls; header configuration: full-width header with logo + nav + actions*. Style and layout are sampled independently, so about a quarter of the fine-grained sites get both and a quarter get neither.

Table 5: Seed components. The last column counts the sites that received a component. *Group-level sites use the first 123 categories and 146 styles.

Component	Pool	Selection	Sites
Capability group	144	2–4 sites each	310
Fine-grained capability	top 2,500 of 19,721	1 site each	2,500
Website category	1,160*	uniform	2,810
Visual style	961*	uniform, $p = 0.5$	1,416
Layout	11 dimensions	composed, $p = 0.5$	1,256

B.4 Website generation and self-validation

Harness. We run Claude Code headless (`claude -p`) with its default system prompt. The generation prompt (Appendix B.6) is the user message. File, search, and shell tools are pre-approved. The environment is given no internet access, so the agent must create all content and assets itself. We serve the site on a local port during generation, so the agent can load and test its pages. A session runs for at most 120 minutes.

Site requirements. Sites are frontend-only: vanilla HTML, CSS, and JavaScript without frameworks or CDN links. State, such as a shopping cart, lives in `localStorage` or `sessionStorage`. Sites must work at 1280×720 and 1920×1080. The prompt asks for realistic content at scale, e.g., at least 100 items in a catalog, with a distinct image for each item.

Tools. The prompt documents three command-line tools.

- *Image generation* with a local Z-Image-Turbo (Team et al., 2025) server (9 inference steps). The tool generates at the supported resolution closest to the requested aspect ratio, then crops and resizes to the requested size. If the crop would remove much of the image, it first adds composition hints to the image prompt.
- *Verification* with Playwright. It checks that the required files exist and that the task files are valid. It also checks that the landing page loads without JavaScript errors, that it has content, clickable elements, a stylesheet, and scripts, and that all internal links resolve.
- *Screenshots* of given pages at both resolutions. The agent reads the images to check the rendering.

Memory. The agent keeps a `CLAUDE.md` file with the site's architecture, features, tasks, known issues, and unfinished work. A restarted session reads this file and continues where the last one stopped.

Stopping criteria. The agent decides when it is done. The prompt asks it to iterate until verification passes, screenshots show no broken layouts or missing images, all buttons and links work, every task is completable, and `CLAUDE.md` is up to date. We then rerun the verification script. If the site is empty or verification fails, we start a new session with the error report appended to the prompt, for at most 10 attempts.

Subagents. Claude Code can delegate work to subagents. The prompt tells the agent when to delegate, and the agent writes the subagent prompts. Besides the mandatory task review (Appendix B.5), agents delegated bulk image generation on 632 sites, e.g., 60 product images per subagent. That said,

Table 6: Layout dimensions. A layout takes one uniformly sampled option from each required dimension and, with probability 0.6, from each optional dimension (†).

Dimension	#	Options
Navigation position	8	top edge; bottom edge; left side; right side; floating center-bottom; floating top-right; full-screen overlay; none (inline/embedded only)
Navigation style	20	horizontal bar with text labels; horizontal bar with icons only; vertical sidebar with icons + labels; vertical sidebar icons-only (rail); hamburger menu expanding to drawer; tab strip with underline indicator; pill-shaped tab buttons; segmented toggle control; breadcrumb chain; mega dropdown with columns; tree/hierarchy nav; command palette / search overlay; ribbon toolbar with grouped actions; pagination dots; anchor links with scroll-spy; flyout cascading submenus; bottom sheet with options; radial/pie menu; chip/tag cloud navigation; step progress indicator
Primary content layout	25	single column centered narrow; single column full-width; two equal columns side-by-side; two columns asymmetric (wide main + narrow aside); three equal columns; four-column grid; responsive card grid (auto-fill); masonry/staggered columns; kanban board columns; dense data table with sortable headers; full-width stacked hero sections; split master-detail (list + detail); dashboard widget tiles; newspaper-style multi-column flow; mosaic/collage tiles; vertical timeline; horizontal timeline; accordion/collapsible panels; tabbed content panels; horizontal carousel/slider; free-form absolute-positioned blocks; centered single-item focus; alternating left-right zigzag sections; nested grid-within-grid; overlay cards on background image
Page model	12	multi-page with shared navigation; single-page application (hash routing); single-page anchor scroll sections; wizard/stepper multi-step flow; infinite scroll continuous feed; modal/dialog-driven (content in overlays); hub-and-spoke (index page + detail pages); slide deck / presentation mode; overlay/drawer panels; paginated listing with page controls; nested routed panels (panels within panels); progressive reveal (content loads on demand)
Navigation behavior†	10	fixed/sticky (always visible); scrolls with page content; collapses to compact on scroll down; auto-hides then reappears on scroll up; expands on hover / collapses on leave; toggleable via dedicated button; reveals on focus/keyboard only; transparent over content then opaque on scroll; slides in from off-screen; fades in/out based on scroll position
Secondary content areas†	15	no secondary areas; right sidebar widgets; left sidebar filters; top banner/announcement bar; bottom persistent bar/dock; floating action button; slide-out info panel; modal dialogs for detail; tooltip-based micro-content; popover cards on hover; inline expandable sections; footer mega-content zone; notification toast area; pinned comparison tray; breadcrumb + contextual toolbar
Content density†	8	ultra-sparse (heavy whitespace, few elements); airy/spacious; moderate balanced; moderately dense; information-dense compact; data-heavy with dense tables; media-rich (images/video dominant); variable density (sparse header, dense body)
Interaction paradigm†	18	standard click navigation links; drag-and-drop reordering; hover-reveal menus/tooltips; search-first (prominent search drives navigation); faceted filter controls (checkboxes/sliders/ranges); inline click-to-edit fields; keyboard shortcut driven; real-time live-updating content; progressive disclosure (expand for more); right-click context menus; bulk multi-select with batch actions; swipe/gesture based; toggle switches and controls; scroll-triggered animations/reveals; form-heavy data entry; comparison/side-by-side selection; undo/redo action flow; voice/accessibility command driven
Header configuration†	10	full-width header with logo + nav + actions; minimal slim header (logo + hamburger); no header (navigation embedded elsewhere); hero/splash header with CTA overlaid; transparent header overlaying hero image; dual bar (utility strip + main nav bar); centered logo header with side nav links; search-focused header (large search input); header with integrated tab navigation; sticky compact header with expandable mega row
Footer configuration†	10	multi-column link footer; minimal single-line footer; no footer; fat footer with newsletter + social + sitemap; sticky bottom bar with key actions; footer with embedded contact form; accordion-style collapsible footer sections; footer as secondary navigation hub; back-to-top + minimal credits; two-tier footer (links row + copyright row)
Sidebar configuration†	8	no sidebars; persistent left sidebar; persistent right sidebar; collapsible left sidebar (toggle); collapsible right sidebar (toggle); dual sidebars (left nav + right detail); overlay sidebar (slides over content); sidebar below breakpoint becomes bottom sheet

for the most part, images were generated in the main agent and we allowed at most three background tasks, because many parallel requests tended to overload the local image server.

B.5 Task and criteria generation

The agent writes the tasks together with the site, so it knows every ground-truth value. Each task is a JSON file with an intent and a list of criteria for the LLM judge (Appendix F.4). The box below shows two examples. Sites have 17.3 tasks on average, and at least 10 are required. WEBLICA-val keeps two tasks per site.

Design rules. The prompt explains how tasks are used. The policy starts from a fresh browser, sees only the intent, and knows nothing about the site. The judge sees only the screenshots of the trajectory. This leads to four rules:

- Intents state a goal and a deliverable without referring to the site's structure, e.g., *Find the cheapest laptop and state its name and price*, not *Go to the boats page and filter by price*.

- Tasks create the state they need, e.g., *Add a laptop to your cart, then remove it*, not *Remove the laptop from your cart*.
- Criteria check both the process (*Applies the brand filter*) and the outcome, with exact expected values (*States the price as '$29.99'*).
- Criteria must be visible in screenshots. Internal state such as `localStorage` cannot be checked.

The number of criteria sets the difficulty: 1–2 for easy, 2–3 for medium, and 3–4 for hard tasks.

Task review. Once the site works, the prompt requires the agent to spawn a subagent that reviews all tasks. The reviewer reads the task files and the site's code. For each task, it checks that (i) the intent is clear without knowledge of the site, (ii) the task assumes no prior state, (iii) every criterion is visible in screenshots, and (iv) the implemented functionality makes the task completable. It also lists features that have no task. The agent then fixes, removes, or adds tasks. Appendix B.6 shows the instructions and a request written by the agent.

Example tasks of a kitchenware site targeting `search_within_combobox`

```
{
  "task_id": 1,
  "intent": "What is the customer support phone number for Chaos Kitchen?",
  "eval": {
    "eval_types": [
      "llm_judge"
    ],
    "criteria": [
      "Navigates to the contact page",
      "States the phone number as '1-800-555-0199' or equivalent"
    ]
  }
}
{
  "task_id": 9,
  "intent": "Use the brand filter to search for brands containing 'Punk'. Select one and
find the most expensive product from that brand.",
  "eval": {
    "eval_types": [
      "llm_judge"
    ],
    "criteria": [
      "Types 'Punk' into the brand combobox search field",
      "Selects 'Punk Provisions' from the filtered dropdown options",
      "Uses the price sort (high to low) button to find most expensive",
      "States the name and price of the most expensive Punk Provisions product"
    ]
  }
}
```

B.6 PROMPTS

Website generation. Below is the template of the website generation prompt, with placeholders in blue. `{domain}` and `{capability}` take the seed, e.g., `kitchenware` and `search_within_combobox`. `{visual_style_section}` and `{layout_section}` take one of the variants in the box *Conditional prompt sections*. The remaining placeholders take paths and ports.

Website generation prompt

```
# Generate a {domain} website with {capability} functionality

You are generating a synthetic website for training web agents. Create a complete,
functional frontend-only website.

## Continuation Mode

**If the directory `{site_path}` already exists**, this is a continuation of previous work:
1. First, read `{site_path}/CLAUDE.md` if it exists to understand:
   - What has been implemented
   - What remains unfinished
   - Any known issues or considerations
2. Continue from where the previous session left off
```

```
3. Focus on completing unfinished work and fixing issues

**If the directory does not exist**, start fresh following the full instructions below.

## Site Configuration
- **Domain**: {domain}
- **Capability**: {capability}
- **Port**: {port}
- **Site name**: {site_name}
- **Output path**: {site_path}

{visual_style_section}

{layout_section}

## Required Output Structure

Create these files in `{site_path}/`:

{site_name}/
|-- index.html              # Main entry page
|-- [additional].html       # Additional pages as needed
|-- css/
|   `-- styles.css          # All styles
|-- js/
|   `-- app.js              # All JavaScript functionality
|-- images/                 # Generated images
`-- tasks/
    |-- task_1.json         # Easy task
    |-- task_2.json         # Medium task
    `-- [task_N.json]       # Additional tasks as appropriate

## Technical Requirements

### Frontend-Only Architecture
- **No backend**: Use localStorage/sessionStorage for state
- **Self-contained**: All resources must be local (no CDN links)
- **Responsive**: Must work at 1280x720 and 1920x1080
- **Accessible**: Semantic HTML, proper labels, ARIA attributes where needed

### JavaScript Patterns
- Use vanilla JavaScript (no frameworks)
- Provide clear visual feedback for all interactions
- Console should be error-free

### CSS Patterns
- Design should be visually distinctive and appropriate for {domain}
- If a visual style was provided above, use it as creative inspiration
- Visible interactive elements with clear affordances
- Consistent color scheme and typography
- Avoid generic, template-like aesthetics - make bold creative choices

## The Capability: {capability}

Implement this capability fully. The website should be a realistic {domain} site that
naturally incorporates {capability}.

Think about what {capability} means in the context of {domain}:
- What pages are needed?
- What interactive elements?
- What user flows?

## Scope

The generated website should be realistic. Be creative in its creation. If there are items
to display, generate a large number of them. Just for illustration, if you generate a
website with books, generate at least 100 items. Don't be lazy in that regard, consider
image and asset generation cheap. Task subagents for this if useful. Don't shy away from
making elements on the website functional, even if they are not directly required for the
main capability. Be ambitious and self-verify as much as possible.

## Task Design (Critical - Read Carefully)

### How Tasks Are Actually Used

A web navigation agent will:
1. Start at `http://localhost:{port}/` with a **fresh browser** (no cookies, no
localStorage, no prior state)
2. See only the **intent** string from the task
3. Navigate and interact based purely on what it observes on screen
4. Have **zero prior knowledge** of the site's structure, page names, or UI layout

A VLM judge will then:
1. Review the **full trajectory of screenshots** from the agent's session
2. Evaluate each criterion by looking at what's **visually present** in those screenshots
3. Determine success/failure based **only** on observable visual evidence

### Writing Good Intents

Intents should be **goal-oriented with clear deliverables**. Study these patterns from real
benchmarks:
```

````
**Good intent patterns:**
- "Find the cheapest [item] available and state its price." → Goal + deliverable
- "Find the earliest [X] from A to B on [date] and state its departure time." → Specific goal + constraint
- "Identify three [things] and describe their characteristics." → Quantity + extraction
- "What are the operating hours of [X]?" → Question format implies deliverable
- "Find the email for [department], not their [other thing]." → Goal + exclusion

**Bad intents** (assume site knowledge or lack deliverable):
- "Go to the boats page and filter by price" → Assumes agent knows page exists
- "Click the blue Apply Filters button" → References specific UI details
- "Add the third item to cart" → Assumes specific ordering
- "Check the settings" → No clear deliverable

**Key principles:**
- Include **what to find AND what to state/extract**
- Add **qualifiers**: "cheapest", "earliest", "at least three", "under $50"
- Add **constraints**: "not including X", "only from Y category"
- Question format ("What is...?") naturally implies a deliverable

### Writing Good Criteria

Criteria verify both **process** (navigation) AND **outcome** (result). Study these patterns:

**Good criteria patterns (process):**
- "Successfully navigates to the [section] page." → Navigation verification
- "Applies the [brand/category] filter." → Action verification
- "Sorts the results by price, lowest first." → State verification

**Good criteria patterns (outcome) examples:**
- "States the price as '$29.99' or equivalent cheapest price." → Specific expected value. Since you are building this website, you will know the exact ground truth values.
- "Identifies 'John Smith' as the contact person." → Exact name when known
- "The product list shows only items with prices below $50." → Visible state
- "The support phone number '1-800-555-0123' is displayed." → Exact value from your site

**Bad criteria** (not verifiable from screenshots):
- "The cart state is updated in localStorage" → Can't see localStorage
- "The filter function was called" → Can't see JS execution
- "States the exact price" → Vague; include the actual expected value

**Key principles:**
- Use **action language**: "Successfully navigates", "Applies filter", "States the price"
- **Include exact expected values** since you built the site and know the answers
- Add **negative criteria** when relevant: "Does not include X"

### Tasks Must Be Self-Contained

Every task starts with a **fresh browser** (empty cart, no login, no filters).

**Bad** (assumes state): "Remove the laptop from your cart" → Cart is empty
**Good** (self-contained): "Add a laptop to your cart, then remove it" → Creates own state

### Task Format

```json
{
  "task_id": N,
  "intent": "Goal with clear deliverable, using qualifiers and constraints",
  "eval": {
    "eval_types": ["llm_judge"],
    "criteria": [
      "Process verification (navigation/actions taken)",
      "Outcome verification (what is visible/stated)"
    ]
  }
}
```

### Task Difficulty Guidelines

Create 10+ tasks with varying difficulty:

**Easy (1-2 criteria):** Single navigation goal
- Intent: "What is the phone number for customer support?"
- Criteria: ["Navigates to a contact or support page", "States the phone number"]

**Medium (2-3 criteria):** Multi-step with filters/forms
- Intent: "Find the cheapest laptop and state its name and price."
- Criteria: ["Navigates to products/laptops section", "Sorts or identifies lowest price", "States name and price"]

**Hard (3-4 criteria):** Complex flows
- Intent: "Create an account with email test@example.com, then add any product to your wishlist."
- Criteria: ["Registration form filled with email", "Confirmation message shown", "Wishlist shows one product"]

## Available Tools

### Image Generation Tool
````

````
A Python CLI tool generates images via the vllm-omni server:

```bash
python tools/generate_image.py \
  --prompt "Modern glass office building with rooftop garden, architectural photography" \
  --width 1200 --height 600 \
  --output "{site_path}/images/hero.webp" \
  --port {image_server_port}
```

Parameters:
- `--prompt` (required): Detailed text description
- `--width`, `--height` (required): Target dimensions in pixels
- `--output` (required): Save path (format inferred from extension: webp, png, jpg)
- `--port` (optional, default 8091): Image server port
- `--seed` (optional): Random seed for reproducibility

Output (JSON to stdout):
```json
{"saved_path": "...", "seed": 1234567890, "generation_size": [1216, 608]}
```

The tool automatically finds optimal generation resolution, applies crop-safe composition
hints when needed, and resizes to exact target dimensions without distortion.

#### Best Practices

- **Write detailed, descriptive prompts.** Image generation prompts should be **as detailed
as possible**, clearly defining the desired content. Include specific details about:
  - Subject matter (what objects, people, or scenes should appear)
  - Style and mood (photography style, lighting, atmosphere)
  - Composition (perspective, framing, focal point)
  - Context and setting (environment, background elements)
  - Etc...
- The tool handles retries automatically for transient errors
- Images are automatically cropped and resized to match your target dimensions without
distortion
- **Generate ALL images the site needs.** Every `<img src="...">` in your HTML and every
`background-image: url(...)` in your CSS MUST have a corresponding generated image file.
Missing images break the site visually and make it look incomplete.
- **One unique image per item.** If there are n products, generate n distinct product
images. If there are m team members, generate m distinct headshots. If there are k gallery
items, generate k unique images. Never reuse the same image for different items.

#### Image Generation (Continuous Process)

Image generation is fast (~1-10s per image, depending on position in queue). Generate
images continuously as you develop the site:

1. **Plan images alongside your site architecture.** When you decide what pages and content
you need, also identify what images each page requires. This happens naturally as part of
site planning--not as a separate step.

2. **Generate images sequentially or in small batches.** Call generate_image.py directly in
foreground Bash calls. You can chain multiple calls with `&&` in a single Bash invocation
(e.g., 5-10 at a time). **Do NOT spawn large numbers of concurrent background tasks** --
this can overwhelm the image server and cause cascading failures. If you use background
tasks at all, limit to at most 3 concurrent.

3. **Generate images as you go.** As you build new pages or add content, generate the
images they need. Don't try to predict everything upfront--just generate what you need when
you need it. Image generation is fast, so treat it as basically free.

4. **Audit before screenshots.** Before taking final screenshots, verify all image
references have files:
   - Check `<img src="...">` references in HTML
   - Check `background-image: url(...)` in CSS
   - Generate any missing images

5. **If image generation fails, retry directly -- never sleep/wait.** If a
generate_image.py call fails, first check the server is reachable with `curl -s
http://localhost:{port}/v1/models`. If the server is down, report the error and move on. If
it's up, retry the failed image immediately in a foreground Bash call. Do NOT use `sleep`
or polling loops to wait for images to appear -- this wastes time and will not fix a
crashed server.

The key principle: **image generation should happen continuously throughout development**,
not just at the start. Every time you add content that needs an image, generate it.

### Verification (Run After Creating Site)
```bash
python {verify_tool} {site_path} {port}
```

### Screenshots (REQUIRED - Visual Verification)

**You MUST take and review screenshots before considering the site complete.** This is how
you verify the site renders correctly.

1. **Capture screenshots** of all main pages at both resolutions:
```bash
````
````

```
python {screenshot_tool} {port} -p / /boats.html /articles.html -o
{working_dir}/screenshots/
```

Replace the page paths with your actual pages (e.g., `/`, `/products.html`, `/about.html`).

2. **View each screenshot** using the Read tool to verify:
   - Layout renders correctly (no broken CSS)
   - Content is visible and properly styled
   - Interactive elements are visible
   - The design looks professional and appropriate for {domain}

3. **Fix any issues** you observe in the screenshots before proceeding.

If the screenshot tool fails with a network error, the site may not be accessible yet or there may be a proxy configuration issue. Verify the site is reachable with `curl http://localhost:{port}/` first.

Process

1. **Plan site architecture** - Decide what pages, functionality, and content you need for {capability} in a {domain} context. As you plan each page, note what images it will need.
2. **Create structure** - Make directories and files
3. **Build iteratively with continuous image generation:**
   - Implement HTML for a page → generate that page's images (sequentially, in foreground)
   - Style with CSS → generate any additional images needed for backgrounds/decorations
   - Add JavaScript functionality → generate images for dynamic content (products, items, etc.)
   - **Keep generating images as you go.** Don't try to plan everything upfront.
4. **Write initial tasks** - Create 10+ task files following the task design guidelines above
5. **Audit images** - Before screenshots, verify all `<img src>` and `background-image: url()` references have files. Generate any missing.
6. **Verify** - Run verification tool and fix any issues
7. **Screenshot & Review** - Take screenshots of ALL pages, READ them to visually verify. Look for missing images (broken image icons), and generate any you missed.
8. **Review tasks with subagent** - See "Task Review" section below
9. **Iterate** - Repeat previous steps as needed, fix issues, create assets, expand missing capabilities, etc., and importantly, self-verify!
10. **Document** - Create/update CLAUDE.md (see below)

Task Review (Required)

After the site is functional and screenshots look correct, **spawn a subagent to review all tasks**. This is critical for quality.

The subagent should:
1. Read all task JSON files in `{site_path}/tasks/`
2. Read the site's HTML/JS to understand what functionality exists
3. For each task, verify:
   - **Intent clarity**: Would an agent with zero site knowledge understand what to do?
   - **No assumed state**: Does the task work from a fresh browser (empty cart, no login, no filters)?
   - **Criteria observability**: Can each criterion be verified from screenshots alone?
   - **Task is completable**: Does the required functionality actually exist in the site?
4. Identify **missing tasks**: Are there implemented features without corresponding tasks?
5. Fix or remove problematic tasks, add missing tasks

Spawn the subagent like this:

```
Use a subagent (inheriting the current model) to review all tasks in {site_path}/tasks/.
The subagent should:

1. Read each task_*.json file
2. Read the site's index.html, other HTML pages, and js/app.js to understand implemented functionality
3. For each task, check:
   - Is the intent clear to someone who has never seen this site?
   - Does it assume any pre-existing state (items in cart, applied filters, logged in)?
   - Are all criteria visually observable in screenshots?
   - Can the task actually be completed with the implemented functionality?
4. List any implemented features that don't have corresponding tasks
5. Output a summary of issues found and fixes needed

Then fix the task files based on the subagent's findings, and add additional tasks if there is functionality that was implemented but is not included in the tasks.
```

CLAUDE.md Documentation

Always create or update `{site_path}/CLAUDE.md` at the end of your session. This file enables another Claude Code session to continue your work seamlessly.

Include the following sections:

```markdown
# {site_name}

## Generation Metadata
Copy this information from `_generation_meta.json` (read it first):
```

```
- Git commit: [from _generation_meta.json]
- Capability: [name] (index: [idx])
- Domain: [name] (index: [idx])
- Visual style: [name or None] (index: [idx or N/A])
- Generated at: [timestamp]

## Overview
Brief description of the site, its purpose, and the capability it demonstrates.

## Architecture
- Pages and their purposes
- Key JavaScript functionality
- State management approach (localStorage keys, etc.)

## Implemented Features
- List of completed functionality
- Notable design decisions and why they were made

## Task Summary
- Number of tasks created
- Difficulty distribution
- Brief description of task types

## Known Issues
- Any bugs or edge cases discovered
- Verification warnings that weren't addressed

## Unfinished Work
- Features planned but not implemented
- Improvements that could be made
- Ideas for additional tasks

## Considerations for Future Sessions
- Technical constraints encountered
- Dependencies between components
- Anything the next session should know before making changes
```

Update this file whenever you make significant changes, ensuring it always reflects the current state of the site.

Metadata: Add a Realistic URL

After the site is complete, you MUST update `{site_path}/_generation_meta.json` to add a `"url"` key with a realistic, invented URL for this website.

The URL should:
- Be a plausible full URL (e.g., `https://www.example-domain.com`)
- Match the **{domain}** industry (e.g., a finance site might be `https://www.summitledger.com`, an agriculture site `https://www.greenfieldagri.com`)
- Sound like a real company or service name -- not generic or obviously fake
- Use an appropriate TLD (`.com`, `.org`, `.io`, `.net`, etc.) based on the domain

Read the existing `_generation_meta.json`, add the `"url"` key, and write it back.

Important Notes

- The server auto-serves the site once the directory exists at {site_path}
- Site will be accessible at `http://localhost:{port}/`
- Make the site visually polished and realistic for {domain}
- Ensure all buttons and links are functional
- Test that each task is actually completable by following the intent
- The site must be completely functional without any server-side code
- **CRITICAL: Generate ALL images.** Every image referenced in HTML or CSS must exist. Generate images continuously as you build--whenever you add content that needs an image, generate it. Before finishing, audit that every `<img src>` and `background-image: url()` has a corresponding file.

Begin

First, check if `{site_path}` already exists:
- **If it exists**: Read `{site_path}/CLAUDE.md` to understand the current state and continue from where the previous session left off.
- **If it doesn't exist**: Create the directory structure at `{site_path}` and implement the site from scratch.

Use subagents (inheriting current model) for context-heavy tasks like implementing complex JavaScript functionality. Always update CLAUDE.md before finishing your session.

Conditional prompt sections and retry addenda

Visual style sampled (replaces `{visual_style_section}`):

```
## Visual Style Inspiration

**Style**: {style_name}
**Description**: {style_description}

This visual style is provided as **creative inspiration**, not a strict requirement. Use it to inform your design choices - colors, typography, layout, textures, and overall aesthetic - to the extent that it makes sense for a {domain} website. If aspects of this style
```

```
conflict with usability, accessibility, or the domain's conventions, prioritize function
over style. The goal is to create a unique, memorable website that stands out visually
while remaining professional and usable.

Be creative and interpret this style in your own way. Avoid creating cookie-cutter,
generic-looking websites.
```

No visual style (replaces `{visual_style_section}`):

```
## Visual Style

No specific visual style was provided. You have **complete creative freedom** to design
this website. Create something unique and visually interesting that fits the domain. Avoid
generic, template-like designs - make bold creative choices with colors, typography,
layout, and visual elements.
```

Layout sampled (replaces `{layout_section}`):

```
## Layout Structure

**Layout**: {layout_name}
**Description**: {layout_description}

Use this layout pattern to organize the site's navigation and content structure. Adapt it
as needed for the {domain} domain -- the layout defines the structural skeleton (navigation
style, page organization, content flow), while the visual style (if any) defines
aesthetics. If aspects of this layout conflict with usability for the specific capability,
prioritize function over strict adherence.
```

No layout (replaces `{layout_section}`):

```
## Layout Structure

No specific layout was provided. You have **complete creative freedom** to choose the
site's navigation pattern, page organization, and content flow. Pick a layout that
naturally fits the domain and capability.
```

Appended on retry if the session produced no site files:

```
## Previous Attempt Failed
No site files were created.
Claude Code output:
{first 500 characters of the session output}

Please create all required files.
```

Appended on retry if verification failed:

```
## Previous Attempt Verification Failed
{first 2000 characters of the verification report}

Please fix the issues above.
```

Task review request written by the agent for the kitchenware site

```
Review all tasks in kitchenware_search_within_combobox_580301/tasks/.

1. Read each task_*.json file
2. Read the site's index.html, about.html, contact.html and js/app.js to understand
implemented functionality
3. For each task, check:
   - Is the intent clear to someone who has never seen this site?
   - Does it assume any pre-existing state (items in cart, applied filters, logged in)?
   - Are all criteria visually observable in screenshots?
   - Can the task actually be completed with the implemented functionality?
4. List any implemented features that don't have corresponding tasks
5. Output a summary of issues found and suggested fixes

Key features that should have tasks:
- search_within_combobox: typing in filter dropdowns to search/filter options
- Category, Brand, Material, Price Range filters
- Pagination (PREV/NEXT buttons)
- Sorting (price low-high, price high-low, A-Z name)
- View mode toggle (grid/list)
- Random product picker
- Add to cart functionality
- Cart modal
- Contact form with subject dropdown
- FAQ accordion on contact page
- About page team info

Report any tasks that need fixing or features that need new tasks.
```

Capability extraction. The remaining boxes show the prompts of Appendix B.2. Besides the text, the extraction request contains up to 15 trajectory screenshots. `{action_summary}` lists up to 20 steps, the first 15 and last 5 of longer trajectories. Each step shows the action and up to two sentences

of the agent's reasoning, and repeated actions are merged. Normalization and grouping share one user prompt and differ in their system prompts.

Capability extraction: system prompt

```
You are analyzing a web agent's trajectory on a website. You receive screenshots from the
trajectory plus the action sequence and context data.

Your job is to:
1. Extract EVERY web interaction capability the trajectory demonstrates -- be exhaustive
2. Provide an open-ended analysis of what happened

A "capability" is a higher-level interaction pattern, NOT a single raw action. For example:
- Clicking a button is NOT a capability by itself
- "Navigate a dropdown menu to find a sub-option" IS a capability
- "Fill a multi-field shipping form" IS a capability
- "Use a date picker to select a date" IS a capability

CRITICAL: Be thorough and list EVERY distinct interaction pattern you observe, no matter
how small. If the agent scrolled through results, that's a capability. If it dismissed a
popup, that's a capability. If it switched tabs, used a filter, typed into a search bar,
selected an option from an autocomplete, navigated a sidebar, read data from a table --
each of these is a separate capability. Do NOT summarize multiple distinct interactions
into one generic capability. It is far better to list too many capabilities than too few.

The examples below are just a SMALL SAMPLE to show the naming style (snake_case,
verb-oriented). You are NOT limited to these -- discover and name whatever you see:

EXAMPLE CAPABILITY NAMES (illustrative only, not exhaustive -- invent new ones freely):
use_search_bar, navigate_dropdown_menu, fill_multi_field_form,
apply_price_filter, use_date_picker, select_from_autocomplete,
interact_with_map, use_pagination, sort_table_column,
expand_accordion_section, handle_modal_dialog, dismiss_cookie_banner,
upload_file, use_slider_control, navigate_breadcrumbs,
fill_login_form, use_multi_step_wizard, select_checkbox_options,
use_shopping_cart, submit_contact_form, use_star_rating,
scroll_to_content, navigate_tab_interface, handle_captcha,
use_sidebar_navigation, select_dropdown_option, add_item_to_cart,
select_product_variant, compare_items, use_calculator_tool,
navigate_image_gallery, play_media, extract_table_data,
use_location_selector, configure_settings, manage_account,
apply_sorting, use_calendar_booking, handle_cookie_consent,
navigate_category_tree, read_and_extract_info, use_autocomplete_address,
switch_language_or_locale, handle_popup_overlay, use_star_rating_filter,
share_content, navigate_footer_links, use_chat_widget

ELEMENT TYPE VOCABULARY (use from this list when applicable):
text_input, search_bar, textarea, password_field,
dropdown, select_menu, multi_select, combobox,
checkbox, radio_button, toggle_switch,
button, submit_button, icon_button, link, tab,
nav_menu, breadcrumb, sidebar_nav, pagination, hamburger_menu,
date_picker, calendar_widget, time_picker,
slider, range_slider, number_stepper,
file_upload, drag_and_drop_upload, image_upload,
modal, dialog, popup, tooltip, toast_notification,
accordion, expandable_section, collapsible_panel,
table, data_grid, sortable_table,
map_widget, interactive_map, map_marker,
carousel, image_gallery, lightbox,
video_player, audio_player,
form, multi_step_form, inline_edit,
filter_panel, faceted_filter, price_filter, category_filter,
shopping_cart, quantity_selector, product_card,
autocomplete, typeahead, suggestion_list,
captcha, recaptcha,
cookie_banner, consent_dialog,
chat_widget, contact_form,
star_rating, review_form,
progress_bar, loading_spinner

For complexity_estimate: "simple" (1-3 steps, one page), "moderate" (4-8 steps or 2-3
pages), "complex" (9+ steps or 4+ pages or multi-step forms)

For website_category, use one of: shopping, travel, utilities, finance, food,
entertainment, social_media, news, education, government, health, real_estate, automotive,
jobs, other

For task_types, use from: search, form_submission, calculator_tool, navigation, comparison,
booking, account_management, information_extraction, configuration, purchasing, other

If the agent was blocked by CAPTCHA, access denial, Cloudflare verification, bot detection,
rate limiting, or the website being unresponsive/timing out, set "website_issue" to one of:
"captcha", "access_denied", "bot_detection", "rate_limiting", "timeout". Otherwise set it
to "none".
```

```
Look broadly and exhaustively for capabilities. Consider all of these categories (include
only what the trajectory actually demonstrates):
- Navigation: menus, breadcrumbs, tabs, sidebars, pagination, back/forward, footer links,
category trees
- Search & filtering: search bars, autocomplete, faceted filters, sorting,
price/date/rating filters
- Forms & input: text fields, dropdowns, checkboxes, radio buttons, date pickers, file
uploads, multi-step wizards, address autocomplete
- Content interaction: accordions, carousels, galleries, lightboxes, tables, maps, media
players, tooltips
- E-commerce: adding to cart, variant selection, quantity changes, wishlists, comparison,
checkout steps
- Account: login, registration, profile management, settings, preferences
- Overlays: modals, popups, cookie banners, chat widgets, notifications, consent dialogs
- Data extraction: reading info from pages, extracting structured data from tables or lists
- Tools: calculators, estimators, booking widgets, scheduling, configurators

For the open_analysis field: Write a descriptive paragraph about what happened in the
trajectory. Describe what the agent did step by step at a high level, what UI elements it
encountered, what worked and what didn't, and any notable aspects of the website's design
or complexity. This should read like a brief case study.

Output ONLY valid JSON with this exact structure:

{"capabilities": [
  {"name": "snake_case_name",
   "label": "Human Readable Label",
   "description": "What this capability involves in context of this task",
   "element_type": "from_vocabulary_above",
   "action_types": ["click", "type", "scroll", "etc"],
   "was_successful": true,
   "confidence": "high|medium|low"}
],
"task_types": ["search", "form_submission"],
"website_category": "travel",
"complexity_estimate": "simple|moderate|complex",
"website_issue": "none",
"open_analysis": "Descriptive paragraph about what happened...",
"action_sequence_summary": "brief arrow-separated summary of key actions"
}
```

Capability extraction: user prompt

```
Task Objective: {objective}

Website: {website}
Starting URL: {starting_url}
Outcome: {outcome}

Pages Visited ({num_urls} unique URLs):
  1. {url_1}
  2. {url_2}
  ...

Action Sequence:
{action_summary}

The screenshots show the website at various points during the trajectory. Analyze them
along with the action data above. Respond with JSON only.
```

Capability normalization: system prompt

```
You are a data-cleaning assistant. You receive a list of capability names (snake_case
strings) extracted from web agent trajectory analyses. Many names describe the same
underlying UI interaction pattern but use different wording (e.g. "navigate_dropdown_menu",
"use_dropdown_navigation", "click_dropdown" all mean the same thing).

Your job:
1. Cluster the names that refer to the SAME interaction pattern.
2. For each cluster, pick ONE canonical name. Prefer the most general, reusable,
verb-oriented snake_case name. Avoid domain-specific names (e.g. prefer "add_item_to_cart"
over "add_sofa_to_cart").
3. Names that are already unique and specific enough should map to themselves.

Output ONLY valid JSON: an object mapping every input name to its canonical name.
Example:
{"navigate_dropdown_menu": "navigate_dropdown_menu", "use_dropdown_navigation":
"navigate_dropdown_menu", "click_dropdown": "navigate_dropdown_menu", "use_search_bar":
"use_search_bar"}

Rules:
- Every input name MUST appear as a key in the output.
- Canonical names must be snake_case.
```

```
- Do NOT invent canonical names that weren't in the input unless merging creates a better
generic name (e.g. "navigate_to_store_page" + "navigate_to_product_page" →
"navigate_to_page").
- When in doubt, keep names separate rather than over-merging.
- Frequency counts are provided to help you prioritize: higher-frequency names are better
canonical candidates.
```

Capability grouping: system prompt

```
You are a data-analysis assistant. You receive a list of fine-grained web interaction
capability names (snake_case) with their frequencies. These come from analyzing web agent
trajectories.

Your job is to map each fine-grained capability to a HIGH-LEVEL functional group. The goal
is to compress ~thousands of specific capabilities into roughly 50-200 broad functional
categories suitable for quantitative analysis.

Guidelines:
- Groups should describe WHAT the agent is doing functionally, not HOW or WHERE.
- Good groups: "form_input", "navigation", "search", "authentication", "cart_management",
"content_filtering", "media_playback", "date_selection", "error_recovery", "accessibility",
"cookie_consent"
- Bad groups (too specific): "click_login_button", "navigate_to_checkout",
"fill_shipping_address"
- Bad groups (too vague): "interaction", "web_action", "general"
- Use snake_case for group names.
- Frequency counts indicate importance -- higher-frequency capabilities should not be
lumped into catch-all groups.
- When unsure, prefer slightly more granular over too vague.

Output ONLY valid JSON: an object mapping every input name to its group name.
Example:
{"navigate_dropdown_menu": "navigation", "fill_email_field": "form_input",
"add_item_to_cart": "cart_management", "dismiss_cookie_banner": "cookie_consent",
"use_search_bar": "search"}

Rules:
- Every input name MUST appear as a key in the output.
- Group names must be snake_case.
- Aim for 50-200 distinct group names total.
```

Capability normalization and grouping: user prompt

```
Here are {N} capability names with their frequencies. Cluster and normalize them. Respond
with a JSON object mapping each name to its canonical form.
  {name_1} (×{count_1})
  {name_2} (×{count_2})
  ...
```

B.7 SEED POOLS

We list all capability groups and samples of the fine-grained capabilities, website categories, and visual styles. Entries appear as they are inserted into the prompt. Capability groups are ordered by the number of fine-grained capabilities per group. The 2,500 fine-grained capabilities span 122 of the 144 groups.

Capability groups (all 144)

```
navigation, content_filtering, data_extraction, scrolling, error_recovery, search,
form_input, results_browsing, waiting_for_load, date_selection, dropdown_interaction,
map_interaction, captcha_handling, ui_state_recognition, menu_navigation, autocomplete,
content_reading, modal_handling, tab_navigation, authentication, error_handling,
external_navigation, content_discovery, media_playback, form_validation, search_refinement,
result_selection, sorting, ad_handling, overlay_handling, access_block_handling,
banner_handling, keyboard_navigation, comparison, location_entry, form_submission,
cart_management, locale_selection, in_page_navigation, content_opening, details_view,
multi_step_flow, media_gallery, accordion_interaction, form_review, pagination,
task_termination, interactive_simulation, variant_selection, url_parameter_management,
editor_tools, url_navigation, location_selection, product_configuration,
carousel_interaction, file_upload, focus_management, cookie_consent,
url_parameter_navigation, hover_interaction, transaction_flow, cta_interaction,
browser_navigation, settings_management, calculator_interaction, workflow_control,
expand_collapse, availability_check, location_search, table_interaction,
chat_widget_handling, checkout_flow, task_constraint_handling, social_interaction,
download_handling, toast_notification_handling, subscription_management,
quantity_selection, empty_state_handling, help_support, promotion_interaction,
```

data_visualization_interaction, newsletter_signup, contact_support, no_results_handling, onboarding, media_error_handling, saved_search_management, slider_interaction, embedded_content_interaction, precision_clicking, tool_usage, time_selection, content_validation, results_view_controls, form_selection_controls, booking_flow, payment, age_verification, list_management, verification_handling, upsell_management, environment_limitation_handling, error_diagnostics, purchase_flow, privacy_consent, feedback_collection, decision_making, content_creation, saving, reviews_interaction, tool_launching, survey_interaction, travel_search, notification_handling, page_refresh, wishlist_management, breadcrumb_navigation, document_viewer, confirmation_handling, alert_handling, seat_selection, context_menu_interaction, quiz_interaction, in_page_find, price_display, site_context_handling, report_generation, fulfillment_selection, tooltip_handling, accessibility, infinite_scroll, traveler_selection, paywall_handling, store_locator, text_interaction, saved_items_management, printing, account_management, geolocation_handling, permission_handling, translation, quote_estimate, drag_and_drop

Fine-grained capabilities (25 samples of 2,500)

browse_recipe_listing_cards, detect_video_playback_error_state, focus_search_input_field, follow_contextual_inline_link, handle_slow_or_stuck_loading_state, navigate_forecast_section_tabs, navigate_to_news_article_page, navigate_to_simulations_catalog, navigate_via_home_logo, navigate_via_sidebar_menu, open_more_sports_menu, open_store_mega_menu, read_dynamic_calculator_output, refine_search_query_iteration, select_address_from_autocomplete, select_model_variant_toggle, select_radio_options_in_modal_form, switch_search_geography_context, use_back_to_top_button, use_custom_price_fields, use_multi_panel_canvas_editor_layout, use_search_bar_as_url_entry, use_search_keywords_field, wait_for_async_content_loading, wait_for_verification_retry, ...

Website categories (25 samples of 1,160)

antiques, bagel_shop, bead_shop, cloud_services, debt_consolidation, dive_center, equestrian, flower_arranging, land_sales, legislative_tracking, magic_shows, mobile_app_development, office_supplies, pet_photography, record_store, river_rafting, skydiving, snowboarding, streetwear, trademark_registration, utilities, watch_repair, wealth_management, wikis, wings_restaurant, ...

Visual styles (25 samples of 961)

Botanical Herbarium Specimen: Detailed pressed flora specimen drawings, botanical label typography, herbarium cream mounts
Card-based Masonry: Pinterest-style staggered cards, varied heights, scannable content blocks
Chalkboard Educational: Dark green/black background, chalk-style hand lettering, classroom nostalgia
Children's Book Illustration: Whimsical drawings, bright colors, large friendly typography, storybook magic
Collage Mixed Media: Layered cut-out images, varied textures, paste-up aesthetic, zine-inspired rawness
Cursor-reactive Interactive: Elements respond to mouse position and movement, playful interactive delight throughout
Dashboard Data Viz: Charts, graphs, metrics displays, information architecture focus, analytics aesthetic
Departure Split-Flap Board: Mechanical flip-board split-flap letter displays, platform numbers, railway timetable grids, precision
Diagonal Angular: Slanted section dividers, dynamic diagonal energy, non-rectangular shapes throughout
Earth Tones: Terracotta, sage green, cream, natural browns, organic warmth, grounded natural palette
Editorial Magazine: Large dramatic headlines, multi-column text layouts, pull quotes, decorative dropcaps, Vogue-style
Fast Food Chain: Bright red/yellow primary colors, bold chunky sans-serif, playful energy, appetite appeal
Gaming Esports: Dark with neon accents, angular aggressive shapes, futuristic fonts, energy and motion, Razer-style
Gothic Romantic: Dark elegance, ornate Victorian details, dramatic atmosphere, romantic darkness
Hand-lettered: Custom brush script fonts, hand-drawn decorative elements, organic imperfections, artisan personality
Letterpress Deep Deboss: Deep ink impression deboss shadows, movable type block texture, cotton paper grain, craft printing
Mac OS Classic: Platinum gray appearance, Chicago bitmap font, black-and-white icons, simple striped title bars
Maximalist Eclectic: More is more, pattern mixing, color clashing, curated chaos, personality explosion
Microinteraction Rich: Elaborate hover states, loading animations, button feedback, interaction design showcase

```
Neon Sign: Glowing tube letter effects, bar signage aesthetic, night city atmosphere
Scandinavian Minimal: Warm whites and creams, natural light wood tones, clean lines, hygge
   cozy simplicity
Streetwear Hype: Bold condensed typography, urban textures, limited edition urgency,
   culture references, Supreme-style
Swiss International Style: Strict grid-based layouts, Helvetica typography, asymmetric
   balance, minimal color palette
Transit Map Beck Diagram: Colored route lines, interchange node circles, Beck-style
   topological simplification, grid-snapped
Wireframe Gray Box Sketch: Gray placeholder boxes, annotation arrow labels, hand-drawn UI
   sketch lines, pre-design prototype
...
```

B.8 Environment generation cost

We compute the cost from the token usage that Claude Code logs for every site, including retries, at public API pricing (Table 7). One environment costs $6.9 on average (median $6.5), and all environments together cost about $19K. This is a one-time cost, since every environment is reused across all RL runs. The estimate excludes the GPU that serves the image model during generation.

Table 7: Generation cost of WEBLICA-SYNTH using Opus 4.5 API pricing.

	Per environment			
	Mean	Median	10th–90th pct.	Total
API cost (USD)	6.9	6.5	4.5–9.5	19.2K
Session time	24 min	23 min	16–34 min	1,108 h

C WEBLICA-CACHE: CONSTRUCTION DETAILS

This section details the construction of WEBLICA-CACHE (Section 3.3), and summarizes how training on it compares with training on synthesized environments.

C.1 Recording and replay

We implement HTTP-level caching using Playwright to record and replay web interactions. During recording, we capture all HTTP traffic and index responses by normalized request signatures. The key challenge is handling volatile parameters (timestamps, session tokens) that change between visits and cause cache misses. We address this with a rule-based normalization system that filters such parameters from URLs, headers, and POST bodies, with domain-specific rules and a multi-level fallback for progressive matching. For each task, Qwen3-VL-32B-Instruct performs 8 parallel rollouts at elevated temperature. We cache the union of all HTTP traffic from both successful and failed attempts, so that each session covers a broader set of interactions than a single trajectory.

C.2 Automated rule generation

Developing caching rules for each website requires analyzing its traffic patterns. We automate this with a pipeline. First, we record browsing sessions, capturing all request parameters without filtering. A subsequent playback reveals cache misses, which we fuzzy-match against the recording to identify which parameters changed across visits. These reports are used to synthesize site-specific caching rules and synthetic responses for non-essential endpoints (e.g., analytics). Generated rules are validated through playback with complete network isolation. Only sessions in which the agent completes the task under cached conditions, as judged by our training judge, are retained. This automated approach captures the full fidelity of real web content, including dynamic layouts and UI interactions, and scales to thousands of domains.

C.3 Construction funnel

InstaV3 (Trabucco et al., 2025) provides web navigation tasks across 146K websites. Many of these websites are unreachable, and many tasks are outdated, have changed criteria, or are too easy to be

useful for RL. We therefore first filter to 38K stable queries. Table 8 summarizes the subsequent funnel: 23.7K queries (62.4%) yield at least one successful teacher rollout, and 15.6K of those (65.8%) remain task-completable under replay with complete network isolation. Every retained environment is therefore verified to be completable from the cache alone.

Table 8: Construction funnel of WEBLICA-CACHE.

Stage	# Tasks	Retained
Stable InstaV3 queries (after removing unreachable websites and outdated or trivial tasks)	38K	–
At least one successful teacher rollout (out of 8)	23.7K	62.4%
Task-completable under replay with complete network isolation	15.6K	65.8%

C.4 OUT-OF-SUPPORT ACTIONS

Because a cached website only contains the recorded traffic, an agent can take actions whose requests were never recorded. Such requests cannot be served, analogous to a server error on the live web. Our training judge labels these episodes as website failures (Appendix F.4), and we discard them from the training batch rather than assigning a reward. Out-of-support exploration therefore reduces the number of usable rollouts but does not corrupt the reward signal.

C.5 TRAINING ON WEBLICA-CACHE

We train on WEBLICA-CACHE with the same SFT initialization, RL recipe, and judge as WEBLICA-8B. Training on cached real websites substantially improves over the base model on all live benchmarks: +6.7 points on Online-Mind2Web (28.6% $\rightarrow$ 35.3%), +11.7 on DeepShop (24.1% $\rightarrow$ 35.8%), and +8.4 on WebTailBench (21.8% $\rightarrow$ 30.2%). It also surpasses the Qwen3-VL-32B-Instruct teacher that recorded the sessions on all three live benchmarks (Table 10), indicating that RL on cached environments improves beyond the recording policy rather than merely distilling it. However, training on synthesized environments performs even better overall (Figure 4b).

C.6 LIMITATIONS

Replay verification makes every retained environment task-completable offline, but it also bounds WEBLICA-CACHE by the recording agent: tasks it cannot solve, and interactions it never performed, are absent. Furthermore, cached websites are static snapshots that do not reflect updates over time, and the overall yield (15.6K of 38K queries) of the existing pools limits how far caching scales. Recorded real-website content may also not be freely redistributable. Synthesized environments avoid these limitations by construction, which motivates WEBLICA-SYNTH as the core of our framework.

D ADDITIONAL QUANTITATIVE RESULTS

D.1 RESULTS ACROSS MODEL SIZES

Figure 6 shows pass@k results across all three model sizes and four benchmarks, with corresponding numerical values in Table 9. Training consistently improves over the base model at all scales and evaluation points.

D.2 COMPARISON WITH THE SFT TEACHER AND WEBLICA-CACHE

Table 10 compares WEBLICA-8B with two references. The first is Qwen3-VL-32B-Instruct, the 4$\times$ larger teacher that generated our SFT data and recorded the cached sessions. The second is training on WEBLICA-CACHE, our proxy for live-web training (Section 3.3). All models use the same agent scaffold, 30 steps per attempt, and judges. WEBLICA-8B outperforms the teacher on all four benchmarks at both pass@1 and pass@8. Training on WEBLICA-CACHE also surpasses the teacher on all three live benchmarks. RL on either environment type therefore improves beyond the model that generated the training data, rather than merely distilling it.

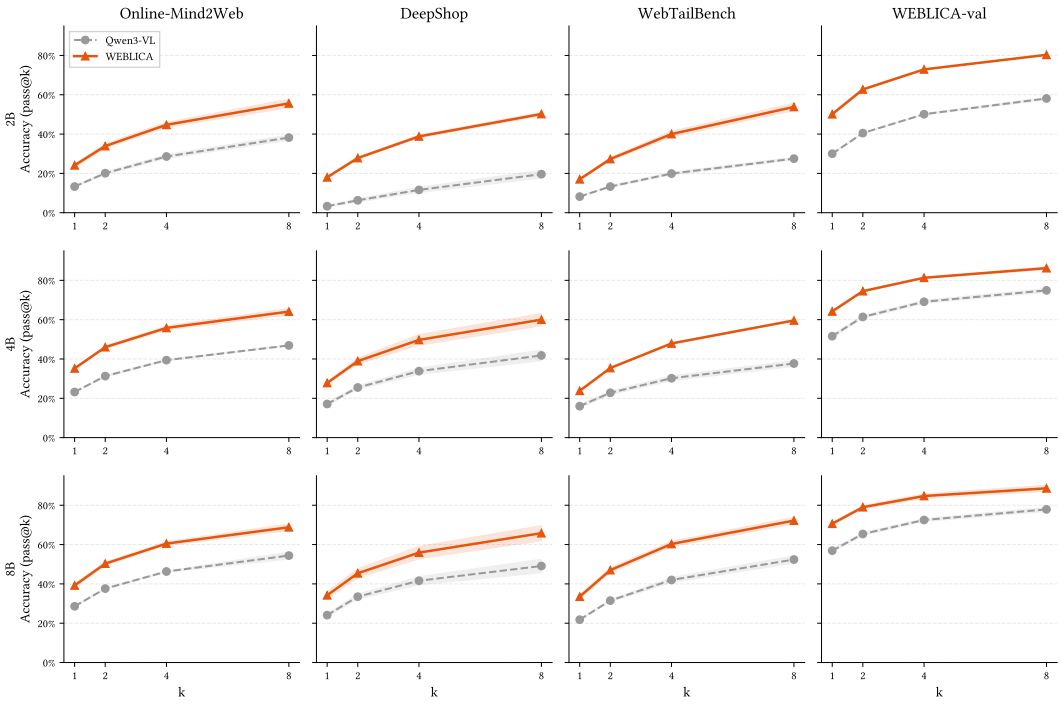


Figure 6: Pass@k accuracy for WEBLICA-2B, WEBLICA-4B, and WEBLICA-8B (orange) compared to their respective Qwen3-VL-Instruct base models (gray) across four benchmarks.

D.3 DOWNSTREAM PERFORMANCE DURING TRAINING

Figure 7 shows steady improvement in pass@1 accuracy on Online-Mind2Web and WEBLICA-val throughout RL training for WEBLICA-8B for a sample training run.

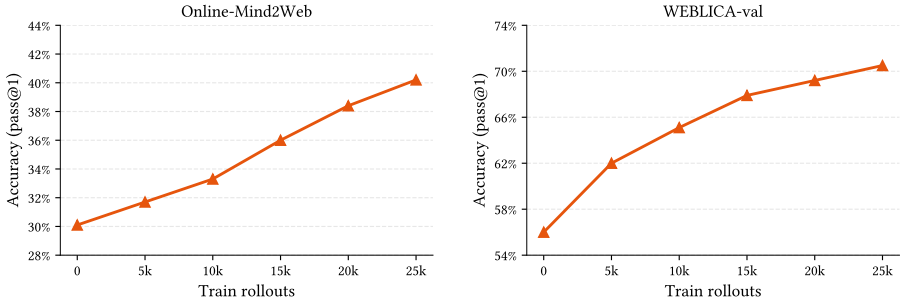


Figure 7: Pass@1 accuracy on Online-Mind2Web (left) and WEBLICA-val (right) during RL training for WEBLICA-8B. Each training step uses 1024 rollouts. The first data point (0 rollouts) is the SFT-initialized model before RL.

Table 9: Pass@k accuracy across model sizes and benchmarks.

		pass@1	pass@2	pass@4	pass@8
Model size: 2B					
Online-Mind2Web	Qwen3-VL	13.3 ± 0.3	20.1 ± 0.6	28.6 ± 0.9	38.2 ± 1.3
	WEBLICA-2B	24.1 ± 0.8	33.9 ± 1.1	44.7 ± 1.2	55.6 ± 2.2
DeepShop	Qwen3-VL	3.3 ± 0.4	6.3 ± 0.8	11.6 ± 1.3	19.6 ± 1.9
	WEBLICA-2B	18.0 ± 0.6	27.8 ± 0.5	38.8 ± 0.7	50.2 ± 0.8
WebTailBench	Qwen3-VL	8.2 ± 0.2	13.3 ± 0.4	19.9 ± 0.7	27.5 ± 0.6
	WEBLICA-2B	17.0 ± 0.4	27.3 ± 0.8	40.0 ± 1.4	53.8 ± 1.9
WEBLICA-val	Qwen3-VL	30.0 ± 0.3	40.5 ± 0.1	50.1 ± 0.3	58.1 ± 0.6
	WEBLICA-2B	50.1 ± 0.6	62.7 ± 0.5	72.9 ± 0.2	80.3 ± 0.2
Model size: 4B					
Online-Mind2Web	Qwen3-VL	23.2 ± 0.2	31.3 ± 0.4	39.4 ± 0.3	46.9 ± 0.2
	WEBLICA-4B	35.2 ± 0.3	46.0 ± 0.6	55.8 ± 1.0	64.1 ± 1.6
DeepShop	Qwen3-VL	17.1 ± 0.5	25.5 ± 0.9	33.8 ± 1.5	41.8 ± 2.7
	WEBLICA-4B	27.8 ± 0.5	38.9 ± 1.5	49.7 ± 2.7	60.0 ± 3.1
WebTailBench	Qwen3-VL	16.0 ± 0.7	22.8 ± 0.9	30.2 ± 1.2	37.7 ± 1.4
	WEBLICA-4B	23.8 ± 0.4	35.4 ± 0.3	47.9 ± 0.3	59.6 ± 0.3
WEBLICA-val	Qwen3-VL	51.6 ± 0.7	61.4 ± 0.7	69.1 ± 0.7	74.9 ± 0.9
	WEBLICA-4B	64.2 ± 0.5	74.5 ± 0.3	81.3 ± 0.2	86.2 ± 0.6
Model size: 8B					
Online-Mind2Web	Qwen3-VL	28.6 ± 0.4	37.6 ± 0.2	46.3 ± 0.3	54.4 ± 1.7
	WEBLICA-8B	39.2 ± 0.9	50.3 ± 0.7	60.5 ± 0.9	68.8 ± 1.7
DeepShop	Qwen3-VL	24.1 ± 0.8	33.5 ± 1.2	41.6 ± 2.0	49.1 ± 3.4
	WEBLICA-8B	34.2 ± 1.7	45.4 ± 2.3	55.9 ± 3.1	65.8 ± 3.8
WebTailBench	Qwen3-VL	21.8 ± 0.2	31.5 ± 0.5	42.0 ± 1.2	52.4 ± 1.8
	WEBLICA-8B	33.5 ± 0.9	47.0 ± 1.1	60.3 ± 1.3	72.2 ± 1.7
WEBLICA-val	Qwen3-VL	56.9 ± 0.7	65.4 ± 0.6	72.5 ± 0.7	77.9 ± 1.0
	WEBLICA-8B	70.6 ± 0.5	79.0 ± 0.7	84.7 ± 1.1	88.6 ± 1.5

Table 10: Comparison with the Qwen3-VL-32B-Instruct teacher and WEBLICA-CACHE (30 steps per attempt). We **bold** the best result per column within each block.

	Model	Online-Mind2Web	DeepShop	WebTailBench	WEBLICA-val
pass@1	Qwen3-VL-8B-Instruct (base)	28.6	24.1	21.8	56.9
	Qwen3-VL-32B-Instruct (teacher)	32.8	33.8	25.2	65.3
	RL on WEBLICA-CACHE	35.3	**35.8**	30.2	63.8
	WEBLICA-8B (RL on WEBLICA-SYNTH)	**39.2**	34.2	**33.5**	**70.6**
pass@8	Qwen3-VL-8B-Instruct (base)	54.4	49.1	52.4	77.9
	Qwen3-VL-32B-Instruct (teacher)	59.0	61.3	56.2	84.6
	WEBLICA-8B (RL on WEBLICA-SYNTH)	**68.8**	**65.8**	**72.2**	**88.6**

D.4 WEBLICA-VAL AND LIVE-WEB PERFORMANCE

Table 11 lists WEBLICA-val and Online-Mind2Web pass@1 for the base and trained models at three sizes and for RL on WEBLICA-CACHE. The two are strongly correlated (Pearson $r = 0.98$, Spearman $\rho = 0.93$ over seven models), and the correlation remains above 0.99 among the four trained models alone, so it is not driven only by the gap between base and trained models. Both benchmarks also improve together during RL training (Figure 7). Although seven models only give a coarse estimate, this suggests that WEBLICA-val can serve as a reasonable offline proxy for live-web performance.

Table 11: Online-Mind2Web and WEBLICA-val pass@1 used for the correlation. Base: Qwen3-VL-Instruct; Ours: WEBLICA models (SFT→RL on WEBLICA-SYNTH); Cache: SFT→RL on WEBLICA-CACHE.

	2B		4B		8B		
	Base	Ours	Base	Ours	Base	Ours	Cache
Online-Mind2Web	13.3	24.1	23.2	35.2	28.6	39.2	35.3
WEBLICA-val	30.0	50.1	51.6	64.2	56.9	70.6	63.8

E ADDITIONAL QUALITATIVE RESULTS

E.1 WEBLICA-SYNTH VISUALIZATIONS

We provide additional random sample screenshots of WEBLICA-SYNTH synthetically generated web environments in Figs. 8 to 10. The generated pages target a broad distribution of web capabilities and are visually diverse.

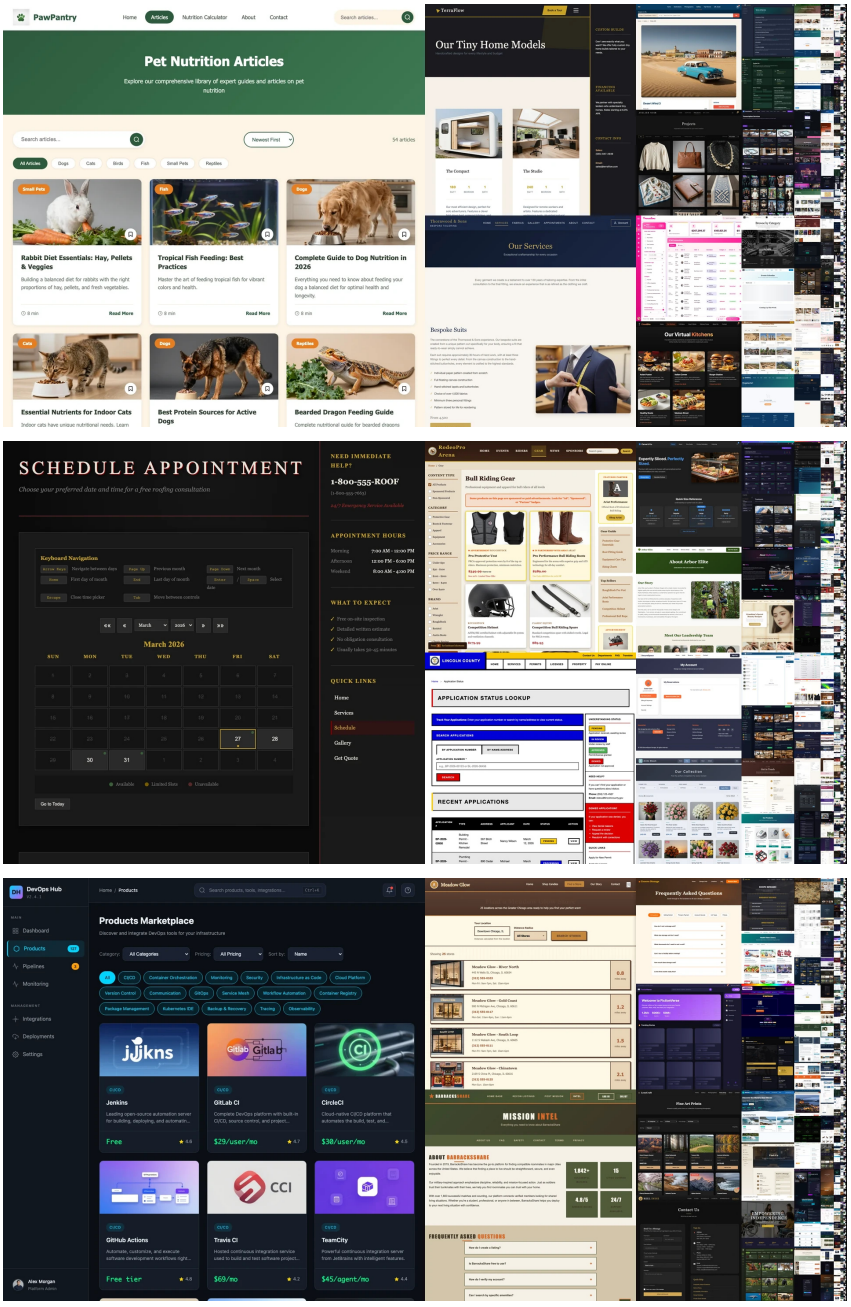


Figure 8: **WEBLICA-SYNTH samples**: Additional visualizations of synthetically generated web environments, capturing a broad set of web capabilities.

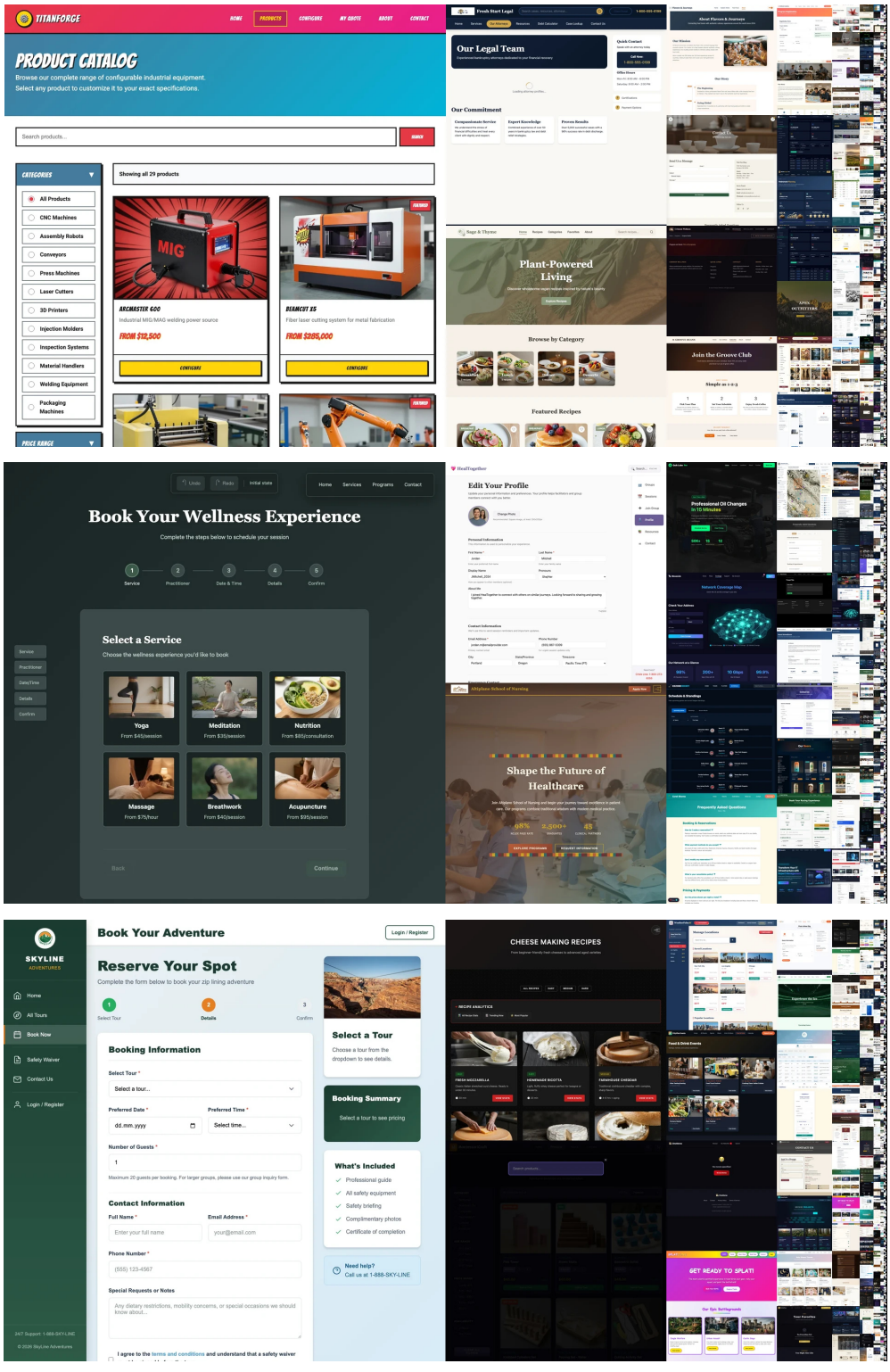


Figure 9: **WEBLICA-SYNTH samples**: Fig. 8 cont.

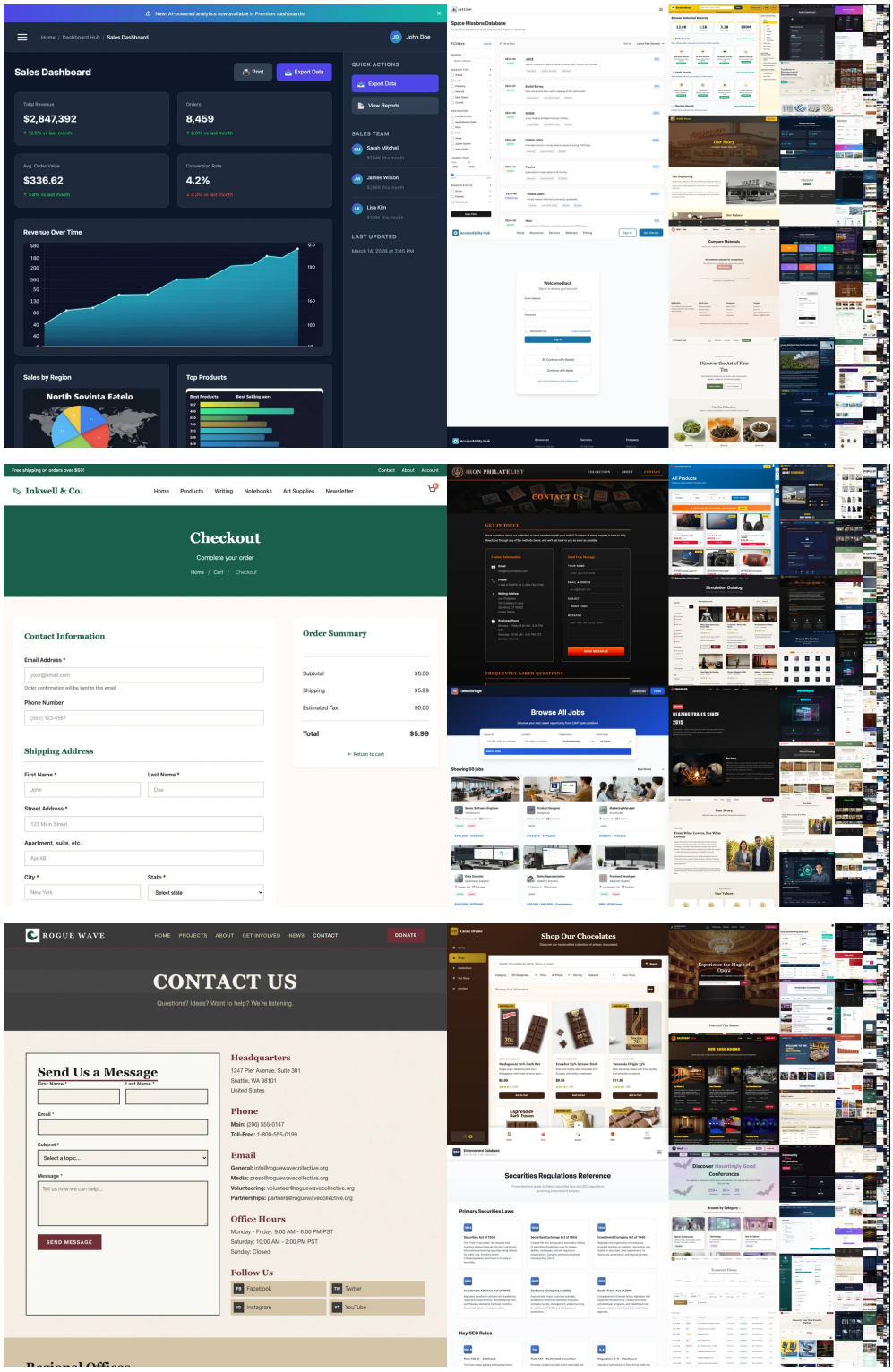
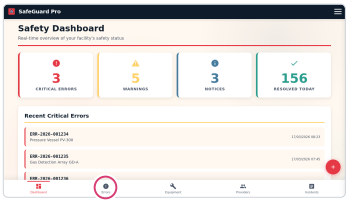


Figure 10: **WEBLICA-SYNTH samples**: Fig. 8 and Fig. 9 cont.

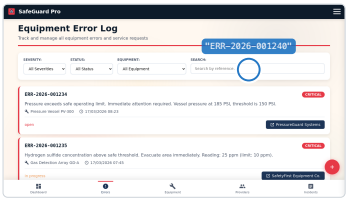
E.2 SOLVED TRAJECTORY EXAMPLES

In Figs. 11 to 17 we show trajectory visualizations of WEBLICA-8B solving WEBLICA-val tasks of various difficulties, and displaying different visual grounding and web navigation skills.

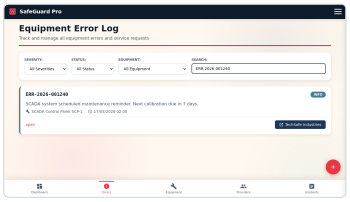
Task: *Search for error reference 'ERR-2026-001240' in the Error Log. What does this error describe and which equipment does it affect?*



Step 1 `click [298, 950]`
Reasoning: *Click the Errors tab to open the error log page*
Criterion 1/4 ✓ → *Navigates to the Error Log page*



Step 2 `type "ERR-2026-001240" Enter`
Reasoning: *Type the error reference to find the matching log entry*
Criterion 2/4 ✓ → *Uses the search function to find ERR-2026-001240*

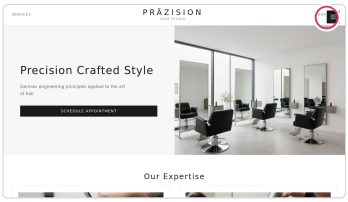


Step 3 `stop [ERR-2026-001240 describes...]`
Reasoning: *Stop after reading the error description and affected equipment*
Criterion 3/4 ✓ → *Identifies it as a SCADA system scheduled maintenance reminder*
Criterion 4/4 ✓ → *States that it affects the SCADA Control Panel SCP-1 equipment*

Result: ✓ Success (4/4 criteria met)

Figure 11: Example trajectory of WEBLICA-8B solving a WEBLICA-val task.

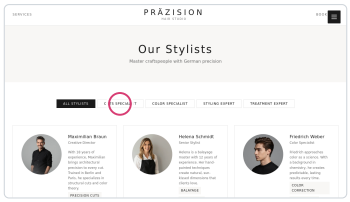
Task: *Which stylist at Präzision Hair Studio specializes in children's cuts? Find their name and list the days they are available.*



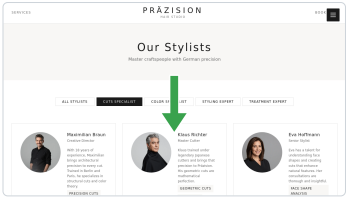
Step 1 `click [945, 65]`
Reasoning: *Click BOOK to open navigation toward stylist information*



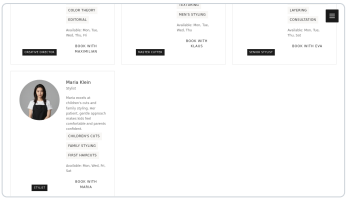
Step 2 `click [67, 706]`
Reasoning: *Click STYLISTS to reach the page listing salon staff*
Criterion 1/3 ✓ → *Navigates to the stylists page*



Step 3 `click [337, 513]`
Reasoning: *Click CUTS SPECIALIST to filter stylists by cutting expertise*



Step 4 `scroll down`
Reasoning: *Scroll down to find a stylist explicitly mentioning children's cuts*

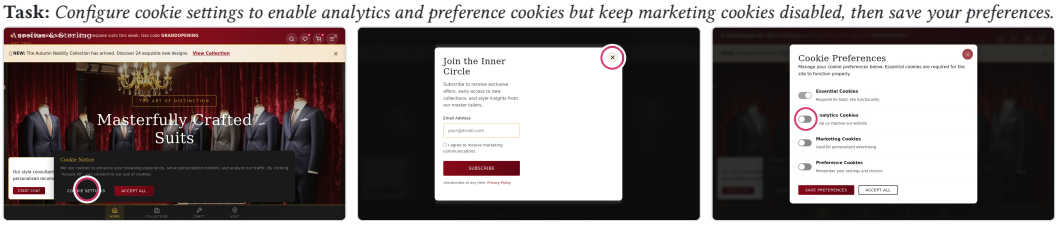


Step 5 `stop [Maria Klein specializes...]`
Reasoning: *Stop after identifying Maria Klein and her available days*
Criterion 2/3 ✓ → *Identifies Maria Klein as the children's cuts specialist*
Criterion 3/3 ✓ → *States her availability as Monday, Wednesday, Friday, and Saturday*

Result: ✓ Success (3/3 criteria met)

Figure 12: Example trajectory of WEBLICA-8B solving a WEBLICA-val task.

Task: *Configure cookie settings to enable analytics and preference cookies but keep marketing cookies disabled, then save your preferences.*



Step 1 `click [244, 840]`
Reasoning: *Click Cookie Settings to open cookie preferences.*
Criterion 1/7 ✓ → *The 'Cookie Settings' button is clicked on the cookie banner*

Step 2 `click [743, 158]`
Reasoning: *Click the close button to dismiss the subscription popup.*

Step 3 `click [273, 475]`
Reasoning: *Click Analytics to enable analytics cookies.*
Criterion 2/7 ✓ → *The cookie settings modal opens*
Criterion 3/7 ✓ → *Analytics cookies toggle is turned ON*

Step 4 `click [273, 720]`
Reasoning: *Click Preference to enable preference cookies.*
Criterion 4/7 ✓ → *Preference cookies toggle is turned ON*

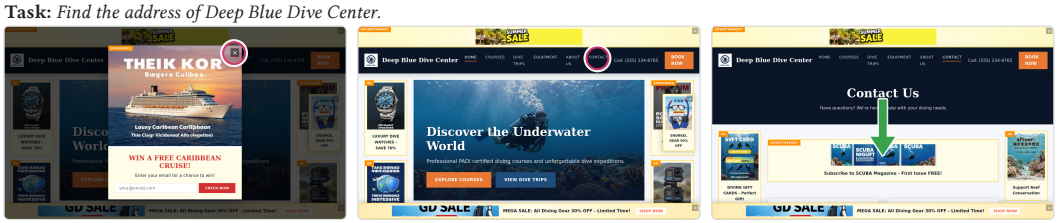
Step 5 `click [334, 840]`
Reasoning: *Click Save Preferences to save the selected cookie settings.*
Criterion 5/7 ✓ → *Marketing cookies toggle remains OFF*
Criterion 6/7 ✓ → *The 'Save Preferences' button is clicked*

Step 6 `stop [Cookie preferences configured & saved]`
Reasoning: *Stop after confirming preferences were saved and the modal closed.*
Criterion 7/7 ✓ → *A success toast appears and the modal closes*

Result: ✓ Success (7/7 criteria met)

Figure 13: Example trajectory of WEBLICA-8B solving a WEBLICA-val task.

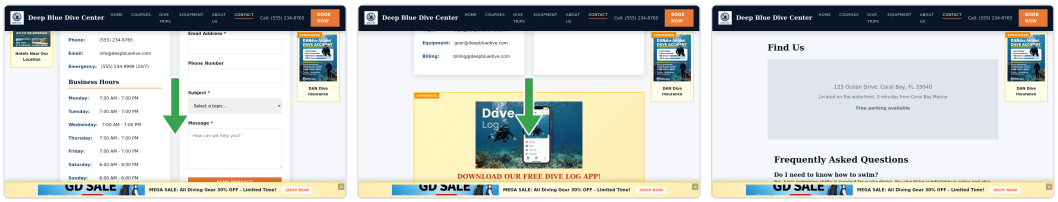
Task: *Find the address of Deep Blue Dive Center.*



Step 1 `click [672, 138]`
Reasoning: *Click the close button to dismiss the popup blocking the site.*

Step 2 `click [700, 165]`
Reasoning: *Click CONTACT to open the page with the business address.*
Criterion 1/2 ✓ → *Navigates to the contact page, about page, or footer section*

Step 3 `scroll down`
Reasoning: *Scroll down to reveal the contact details section.*



Step 4 `scroll down`
Reasoning: *Scroll down to continue searching for the physical address.*

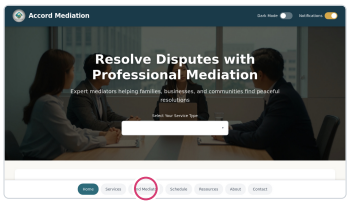
Step 5 `scroll down`
Reasoning: *Scroll down to reach the Find Us section near the bottom.*

Step 6 `stop [123 Ocean Drive, Coral Bay, FL 33040]`
Reasoning: *Stop after locating the address in the Find Us section.*
Criterion 2/2 ✓ → *States the address as '123 Ocean Drive, Coral Bay, FL 33040' or equivalent*

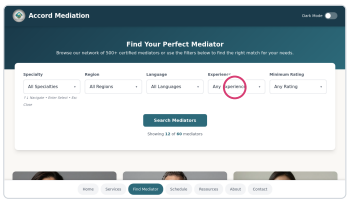
Result: ✓ Success (2/2 criteria met)

Figure 14: Example trajectory of WEBLICA-8B solving a WEBLICA-val task. This synthetic website was designed to contain substantial visual clutter, such as popups and ad banners, yet the agent successfully navigates the page.

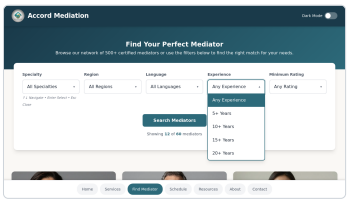
Task: *Using the keyboard, navigate to the Find Mediator page and use the dropdown filters to search for mediators with 15+ years of experience. How many results match?*



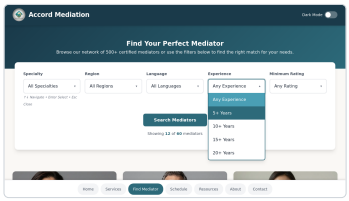
Step 1 `click [414, 950]`
Reasoning: *Click Find Mediator to open the mediator search page*
Criterion 1/5 ✓ → *Navigates to the Find Mediator (mediators.html) page*



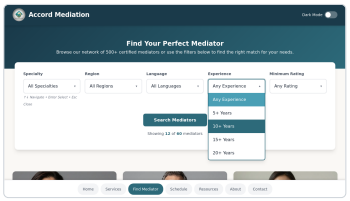
Step 2 `click [675, 425]`
Reasoning: *Click the Experience dropdown to focus and open filter options*



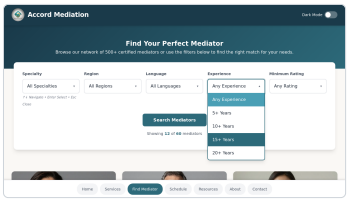
Step 3 `press [ArrowDown]`
Reasoning: *Press ArrowDown to move through experience options*
Criterion 2/5 ✓ → *Uses keyboard navigation to access the Experience dropdown*



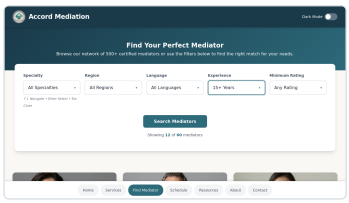
Step 4 `press [ArrowDown]`
Reasoning: *Press ArrowDown to continue toward 15+ Years*



Step 5 `press [ArrowDown]`
Reasoning: *Press ArrowDown to highlight 15+ Years*



Step 6 `press [Enter]`
Reasoning: *Press Enter to select 15+ Years and apply the filter*
Criterion 3/5 ✓ → *Selects '15+ Years' from the experience filter*
Criterion 4/5 ✓ → *Clicks or activates the Search Mediators button*

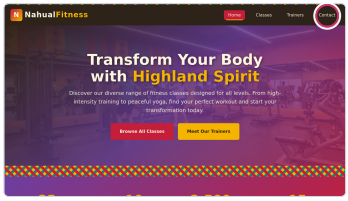


Step 7 `stop [12]`
Reasoning: *Stop after reading that 12 mediators match the criteria*
Criterion 5/5 ✓ → *States the number of mediators matching the criteria*

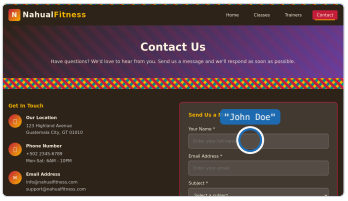
Result: ✓ Success (5/5 criteria met)

Figure 15: Example trajectory of WEBLICA-8B solving a WEBLICA-val task. Although the agent could select the correct dropdown entry directly via clicking, WEBLICA-SYNTH tasks are designed to train web navigation capabilities broadly; in this case, the task explicitly requires completion through keyboard navigation.

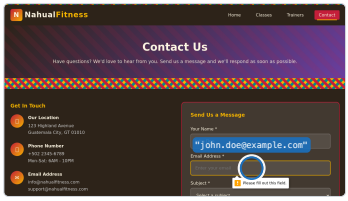
Task: *Send a message to Nahual Fitness asking about membership pricing. Use the name 'John Doe' and email 'john.doe@example.com'.*



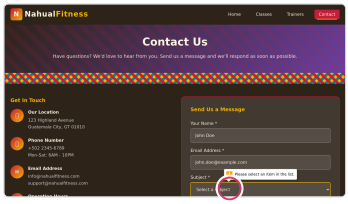
Step 1 `click [946, 62]`
Reasoning: *Click Contact to open the page with the inquiry form*
Criterion 1/6 ✓ → *Navigates to the Contact page*



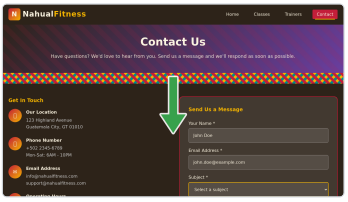
Step 2 `type "John Doe" Enter`
Reasoning: *Type the name to identify the sender*
Criterion 2/6 ✓ → *Fills in name as 'John Doe'*



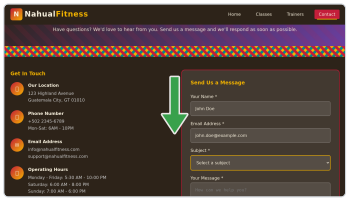
Step 3 `type "john.doe@example.com" Enter`
Reasoning: *Type the email to provide a reply address*
Criterion 3/6 ✓ → *Fills in email as 'john.doe@example.com'*



Step 4 `click [660, 957]`
Reasoning: *Click the subject dropdown to choose an inquiry topic*



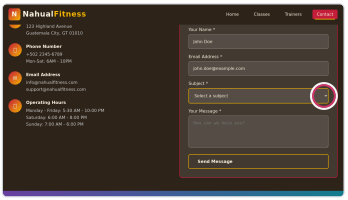
Step 5 `scroll down`
Reasoning: *Scroll to find the relevant subject option*



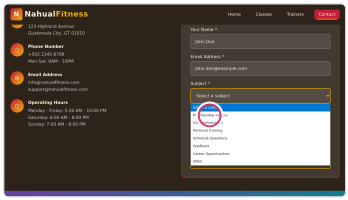
Step 6 `scroll down`
Reasoning: *Scroll to continue locating the correct subject option*



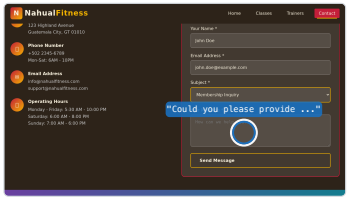
Step 7 `scroll down`
Reasoning: *Scroll to reveal the membership-related subject option*



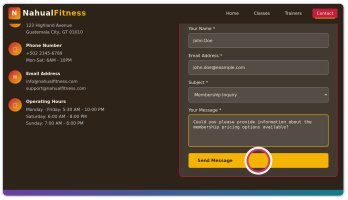
Step 8 `click [941, 477]`
Reasoning: *Click the dropdown to display available subject choices*



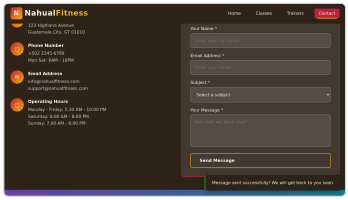
Step 9 `click [602, 578]`
Reasoning: *Click Membership Inquiry to set the subject*
Criterion 4/6 ✓ → *Selects 'Membership Inquiry' as the subject*



Step 10 `type "Could you please provi..." Enter`
Reasoning: *Type a message asking about membership pricing*
Criterion 5/6 ✓ → *Enters a message about membership pricing*



Step 11 `click [749, 816]`
Reasoning: *Click Send Message to submit the completed form*

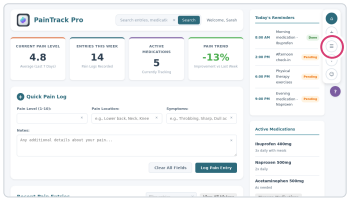


Step 12 `stop [Message sent successfully!]`
Reasoning: *Confirm the success message to verify submission*
Criterion 6/6 ✓ → *Submits the form and sees success confirmation message*

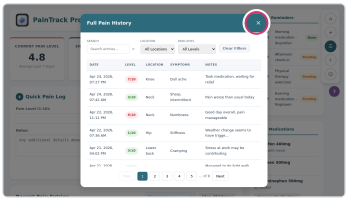
Result: ✓ Success (6/6 criteria met)

Figure 16: Example trajectory of WEBLICA-8B solving a WEBLICA-val task. The agent attempts to open the dropdown menu but, as it lies near the bottom of the page, must first scroll down to reveal it.

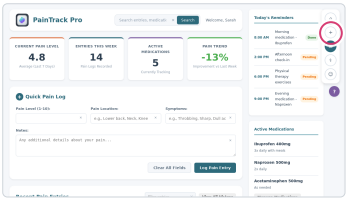
Task: *Search for 'Ibuprofen' in the Medications section. Clear the search and then add a new medication: name 'Meloxicam', dosage '15mg', frequency 'Once daily with food'. Save the medication.*



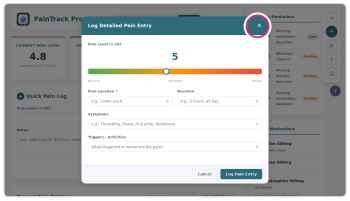
Step 1 `click [959, 222]`
Reasoning: *Click the menu icon to try opening navigation to Medications*



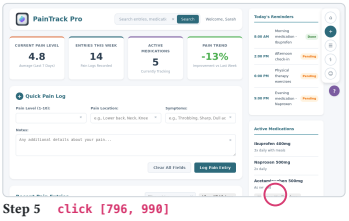
Step 2 `click [741, 98]`
Reasoning: *Click the close button to dismiss the unrelated history dialog*



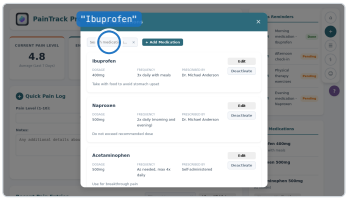
Step 3 `click [959, 150]`
Reasoning: *Click the side plus icon to try accessing medication tools*



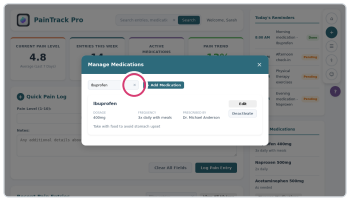
Step 4 `click [741, 117]`
Reasoning: *Click the close button to dismiss the unrelated pain entry modal*



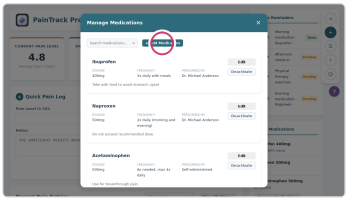
Step 5 `click [796, 990]`
Reasoning: *Click Manage Medications to open the Medications section*
Criterion 1/9 ✓ → *Navigates to the Medications section*



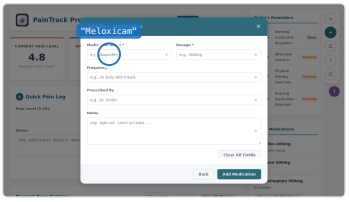
Step 6 `type "Ibuprofen" Enter`
Reasoning: *Type Ibuprofen to search the medications list*
Criterion 2/9 ✓ → *Enters 'Ibuprofen' in the search field*



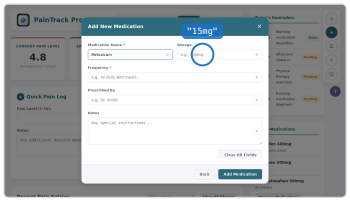
Step 7 `click [380, 420]`
Reasoning: *Click the clear icon to reset the search field*
Criterion 3/9 ✓ → *Clears the search field*



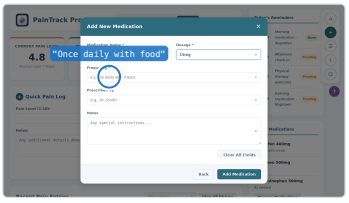
Step 8 `click [465, 204]`
Reasoning: *Click + Add Medication to open the new medication form*
Criterion 4/9 ✓ → *Clicks '+ Add Medication' button*



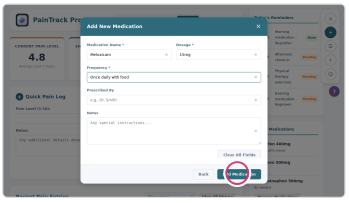
Step 9 `type "Meloxicam" Enter`
Reasoning: *Type Meloxicam to enter the medication name*
Criterion 5/9 ✓ → *Enters 'Meloxicam' as medication name*



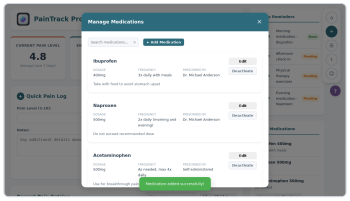
Step 10 `type "15mg" Enter`
Reasoning: *Type 15mg to enter the dosage*
Criterion 6/9 ✓ → *Enters '15mg' as dosage*



Step 11 `type "Once daily with food" Enter`
Reasoning: *Type Once daily with food to enter the frequency*
Criterion 7/9 ✓ → *Enters 'Once daily with food' as frequency*



Step 12 `click [685, 875]`
Reasoning: *Click Add Medication to save the new medication*
Criterion 8/9 ✓ → *Clicks 'Add Medication' to save*



Step 13 `stop [Medication added successfully!]`
Reasoning: *Stop after confirming the medication was added successfully*
Criterion 9/9 ✓ → *Success message confirms medication was added*

Result: ✓Success (9/9 criteria met)

Figure 17: Example trajectory of WEBLICA-8B solving a WEBLICA-val task. The agent initially opens two incorrect menus before identifying *Manage Medications* at the bottom of the screen and successfully editing the medication.

F ADDITIONAL TRAINING DETAILS

F.1 TRAINING HYPERPARAMETERS

We performed a lightweight hyperparameter exploration for both SFT and RL stages. Tables 12 and 13 list the values explored.

Table 12: SFT hyperparameters. **Bold** values indicate selected configurations that performed similarly.

Hyperparameter	Values
Base model	Qwen3-VL-8B-Instruct
Trainable components	Language model only (vision tower + projector frozen)
Learning rate	1e-5, **1e-6**
LR schedule	Cosine
Warmup ratio	0.1
Epochs	1, **2**, **3**
Image resolution	640×360, **1280×720**
Cutoff length	80,000
Effective batch size	8
Precision	bf16

Table 13: RL hyperparameters. **Bold** values indicate selected configurations that performed similarly.

Hyperparameter	Values
Base model	Qwen3-VL-8B-Instruct, **SFT model**
Algorithm	Dr. GRPO
Learning rate	**1e-5**, **5e-6**, 1e-6
LR schedule	Constant
KL coefficient	**0**, **0.005**, **0.01**, 0.02, 0.03, 0.05
Batch size	64, **128**, **256**
Rollouts per prompt (n)	**4**, **8**, 16
Rounds (R)	15, **25**, 35
PPO mini-batch size	32
Max grad norm	1.0
Precision	bf16
Image resolution	1280×720
Max model context length	80,000
Max new tokens per action	512

F.2 Compute

All SFT and RL experiments are conducted on 8 NVIDIA B200 GPUs. Environments are served locally on the same node. Each RL training step generates 1024 rollouts (256 prompts × 4 rollouts per prompt), each with up to 25 interaction rounds, completing in approximately 28 minutes with our local environment setup.

F.3 Action space

Table 14 lists the full action space used by our visual web agent.

Table 14: Action space of visual web agent. Coordinate-based actions take pixel positions as arguments.

Action	**Description**
`click(x, y)`	Click at pixel coordinates
`hover(x, y)`	Hover at pixel coordinates
`type(text, [x, y], [enter])`	Type text, optionally at coordinates and press enter
`press(key)`	Press a keyboard key
`scroll(direction, [amount])`	Scroll in a given direction
`go_back()`	Navigate back in history
`go_forward()`	Navigate forward in history
`wait()`	Wait for page to load
`stop(response)`	Submit response and end episode

F.4 TRAINING REWARD JUDGE PROMPT

During RL training, we use a VLM-based judge (GPT-4o) to evaluate whether the agent successfully completed each task. The judge receives the full trajectory (screenshots and actions) and evaluates against task-specific criteria. Samples labeled as "website failure" are discarded from the training batch. The prompt is shown below.

Training Judge Prompt

System:

```
You are an expert evaluator for web navigation tasks.

Analyze the provided trajectory steps and determine if the agent
successfully completed the following task: {task}

Respond with one of the following based on the task-specific criteria
below:
- 'correct' -- The agent successfully completed the task
- 'incorrect' -- The agent failed to complete the task due to its own
mistakes
- 'website failure' -- The agent was making reasonable progress but was
blocked by technical issues beyond its control

Task-specific evaluation criteria: {criteria}

Technical issues that qualify for 'website failure':
- Page timeouts or loading failures
- Blank or empty pages that fail to render
- Connection errors or server errors (5xx responses)
- CAPTCHA or bot detection blocking the agent
- Pages stuck in infinite loading states
- Elements that fail to become interactive despite multiple attempts

To determine if a website issue occurred, look for these indicators in
the trajectory:
- The agent repeatedly tries the same reasonable action without success
- Screenshots show loading spinners, error messages, or blank content
- The agent uses 'wait' actions multiple times without page progress
- The agent's actions are correct but the page state doesn't change as
expected

Important: Only use 'website failure' when the agent was on a
reasonable path toward completing the task. If the agent made
fundamental mistakes before encountering technical issues, respond
with 'incorrect'.


These are the actions the agent can take:
[Action space description omitted for brevity; see Table 14]
```

User:

```
Step 1 - Screenshot: [image]
Step 1 - Agent Action: [action]
Step 2 - Screenshot: [image]
Step 2 - Agent Action: [action]
...
Based on these trajectory steps, did the agent successfully complete
the task? First respond with your decision followed by your
reasoning.
```

F.5 JUDGE VALIDATION

We validate our training judge in two ways. First, on 100 training tasks, its decisions agree with human labels in 88% of cases. Second, we extend this validation to real websites by collecting 300 trajectories on Online-Mind2Web from 100 tasks and three different agents, and comparing judge decisions with human labels. As shown in Table 15, GPT-4o agrees closely with human labels, and open-weight judges perform on par. The reward therefore does not depend on a proprietary model, at

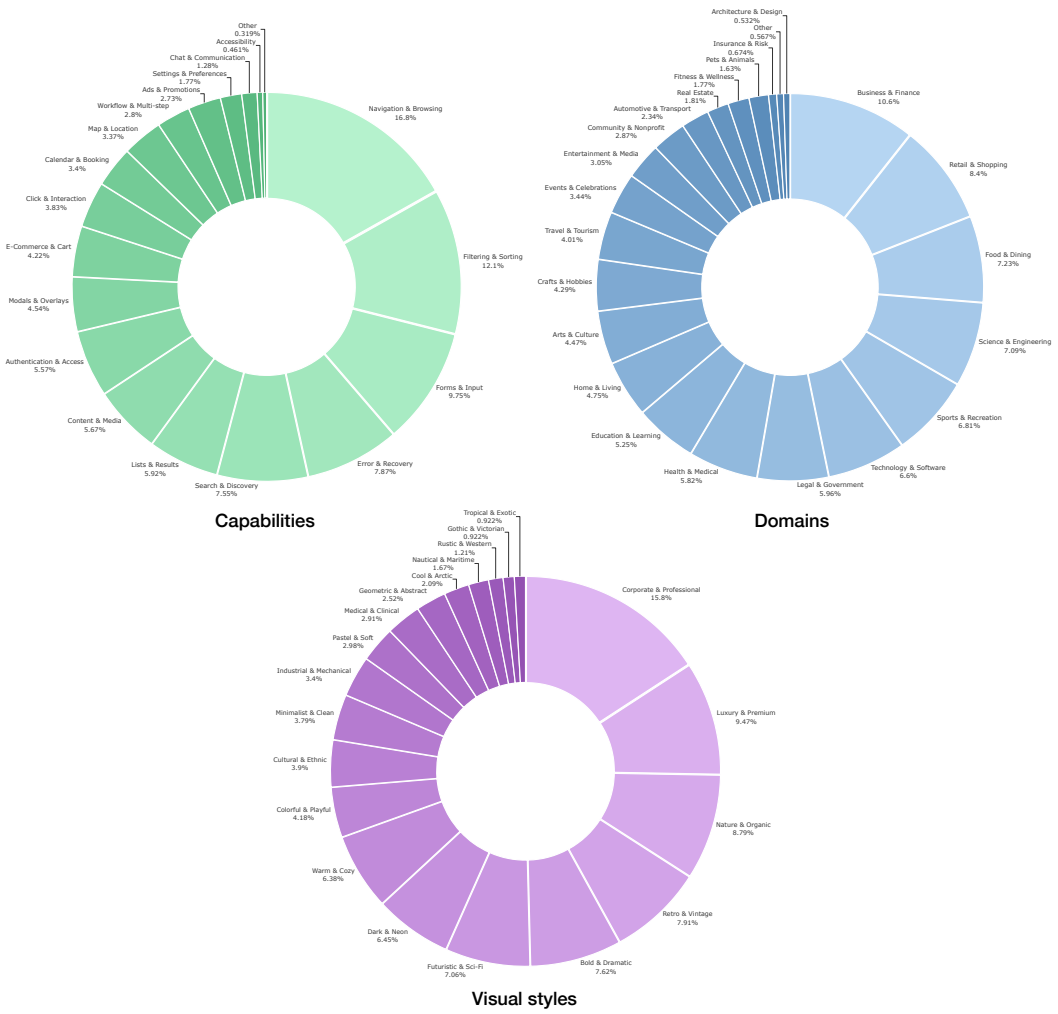


Figure 18: **WEBLICA-SYNTH site grouping**: Distribution of synthetic web browsing tasks grouped across capability categories (top left), domain categories (top right), and visual style categories (bottom).

the cost of additional compute for hosting the judge. Please note that all live-benchmark evaluations in Section 5 use benchmark-specific official judges rather than our training judge.

Table 15: Judge agreement with human labels on 300 Online-Mind2Web trajectories (100 tasks, 3 agents).

Judge	Accuracy	F1	Precision	Recall
GPT-4o	91.0	88.5	84.6	92.9
Qwen3.5-397B	92.7	90.6	86.9	94.6
Qwen3-VL-32B-Instruct	90.6	88.5	81.8	96.4

G DATA STATISTICS

Environments. Figure 18 shows the distribution of synthesized environments across capability, domain, and visual style categories.

SFT Trajectories. Figure 19 shows the distribution of trajectory lengths across the 51.7K SFT training trajectories.

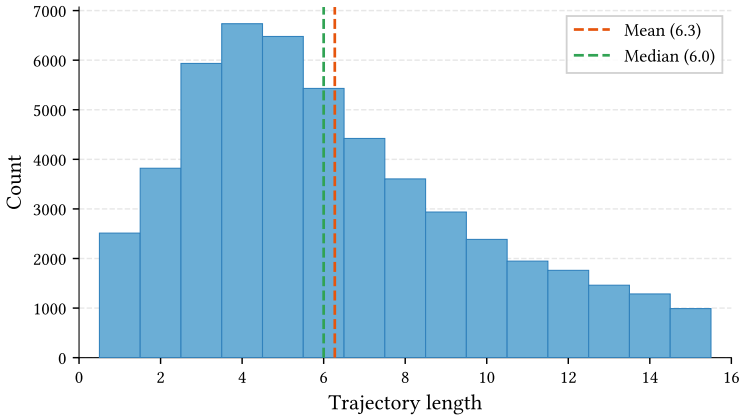


Figure 19: Distribution of trajectory lengths in the SFT training data.