WebGym: Scaling Training Environments for Long-Horizon Visual Web Agents with Realistic Tasks

Supplementary Material

A. Figure Details

Figure 1 (Teaser). Tasks in prior large-scale training setups were relatively simple, e.g., Test-Time-Interaction (TTI), resulting in failures on many held-out tasks. The task shown in Row 1 is: *"From ArXiv, access the website of the university that maintains and manages ArXiv. How many undergraduate students are currently at the university?"*, where the agent fails to access the Cornell website. We build *WebGym* (Row 2, bottom), a significantly larger training environment supporting harder and more diverse tasks. The task shown in Row 2 is: *"Find the product code for the 'Austral Oak TrueScale' laminate benchtop design on laminex.com.au."*, where the base model (untrained) agent gets stuck repeating its behavior (*repeat action*), while the agent trained with *WebGym* (*RL w/ WebGym*) successfully solves the task.

Figure 8 (Ablations). *(left)* Test-set success rate curves of different models (Qwen3-VL-Instruct-8B, Qwen3-VL-Thinking-8B, GPT-4o, and GPT-5-Thinking), and of Qwen3-VL-Instruct-8B under different constraints (either removing the memory prompt or removing the repetition penalty during RL). *(right)* Test-set same-screenshot (repetitive inefficient action) rate and trajectory length curves before and after applying the action repetition penalty (filtering) on the Instruct model. The orange curve is an average of two runs, while the translucent orange area denotes the variance. All plots use EMA smoothing of 0.50.

Figure 9 (Scaling). Test-set success rate curves under different variations to training: removing domains ("exclude domains"), tuning difficulty ratios ("uniform sampling", "biased to hard", "only easy", "only medium"), and shortening the train-time step budget ("shorten horizon"). Results are reported for (a) Overall (all tasks), (b) Easy (1–3), (c) Medium (4–6), and (d) Hard (7+). The orange curve averages two runs; the translucent orange band shows their variation. All plots use EMA smoothing of 0.50.

B. Conclusion, Discussion and Future Work

We introduced *WebGym*, the largest open-source training environment for visual web agents, featuring procedural task construction with structured evaluation rubrics and an asynchronous rollout system that eliminates synchronization barriers. With these design choices, a simple REINFORCE algorithm enables strong generalization to entirely unseen websites, outperforming proprietary models.

Our experiments reveal several empirical insights: explicit memory mechanisms are essential for long-horizon tasks requiring cross-step information retention; penalizing repeated ineffective actions improves sample efficiency; domain diversity directly translates to out-of-distribution generalization; uniform difficulty sampling outperforms both easy-only and hard-biased curricula; and shorter training horizons improve both efficiency and final performance.

Despite the availability of meaningful task-specific rubrics in our task set, LLM-generated rubrics can sometimes be overly strict, which slightly reduces sample efficiency during training. An important direction for future work is to explore more advanced designs for rubric-based evaluation. One promising approach is to treat LLM-generated rubrics as soft guidance rather than strict criteria, for example by training an evaluator that jointly conditions on both the rubric and the task instruction. Another promising direction is to investigate reinforcement learning algorithms that utilize *WebGym* more efficiently, particularly methods based on dynamic curricula [34], automatic curriculum learning [3], multi-agent training, and cross-site navigation.

C. Additional Task Decomposition Example

Take a harder task for example, such as: "*Find an eligible upcoming piano concert in the US or Canada by a Chopin Competition prize winner, providing the pianist's name, competition year/prize, YouTube link to their final performance, and the concert's date, city, venue, and event page link.*" We decompose this into three fact groups: G1 (concert eligibility) with 2 facts (timing and location constraints), G2 (pianist details) with 3 facts (name, competition prize, YouTube link), and G3 (concert details) with 4 facts (date, city, venue, event link), yielding a total difficulty of 9. Since the decomposition conditions are met (3 groups ≥ 2, and G2 and G3 are "large" with $\geq$3 facts each), we generate valid subset combinations: $\{G2\}$ (d=3), $\{G3\}$ (d=4), $\{G1,G2\}$ (d=5), $\{G1,G3\}$ (d=6), and $\{G2,G3\}$ (d=7). The combination $\{G1\}$ alone is invalid since G1 has only 2 facts (no large group), and $\{G1,G2,G3\}$ is excluded as it equals the original task.

D. System Architecture Details

The **server** simulates the browser environments following a master/worker paradigm [1], with a master node routing API requests to CPU worker nodes that host the actual simulators and execute actions. The master node configuration determines how much parallel work the system can support, making it easy to scale horizontally by adding workers or vertically by increasing per-node resources. The server

is entirely CPU-based and requires no GPUs. The **client** hosts agent instances on the GPU and sends requests to the server. To manage the requests sent to the server, we adapt an operation-specific local queue that enables spreading CPU and GPU workload evenly under tight resource settings, with more details in §I.1. The client is GPU-heavy and relies little on CPU budgets.

E. Main Results Table

	Base Model	Prompt	RL on WebGym	Perf. (%)
Proprietary	`GPT-4o`	Memory	✗	27.1
	`GPT-5`	Memory	✗	29.8
Open-source	`Qwen3-VL-Instruct-8B`	Official*	✗	27.4
	`Qwen3-VL-Instruct-8B`	Memory	✗	26.2
	`Qwen3-VL-Thinking-8B`	Memory	✗	28.2
	`Qwen3-VL-Instruct-8B`	Memory	✓**(Ours)**	**42.9**

Table 2. **Main results on the OOD test set.** RL on *WebGym* improves Qwen3-VL-Instruct-8B to 42.9%, surpassing all proprietary and open-source baselines. *Official prompt from QwenLM-Team [31].

F. Policy Update Formulation

We run REINFORCE [38] with binary terminal rewards. The training objective is:

$$\arg\max_{\theta} \; \mathbb{E}_{\mathcal{T}\sim\text{tasks}}\Big[\mathbb{E}_{o_{0:\tau},\, a_{0:\tau-1}\sim\pi_\theta(\cdot|\mathcal{T})} \Big[\Big(\sum_{t=0}^{\tau-1}\log\pi_\theta(a_t \mid o_{\le t},\mathcal{T})\Big)\cdot \mathbb{1}[R(o_{0:\tau},\mathcal{T})=1]\Big]\Big], \quad (1)$$

where θ denotes the parameters of the (multi-step) agent policy π_θ, and $\mathcal{T} \sim$ tasks is a task instance sampled from the training task distribution. A rollout proceeds in discrete interaction steps indexed by t. At each step t, the agent samples an action $a_t \sim \pi_\theta(\cdot \mid o_{\le t}, \mathcal{T})$ conditioned on the full observation history $o_{\le t}$ (and the task), and the environment returns the next observation. The rollout terminates at step τ only when one of the following two conditions are met: **(i)** when the agent emits an `ANSWER` action, or **(ii)** when the step index reaches the training horizon h_{train}; thus $\tau \le h_{\text{train}}$. The binary terminal reward $R(o_{0:\tau}, \mathcal{T}) \in \{0, 1\}$ indicates whether the overall trajectory succeeds on task $\mathcal{T}$. The objective increases the log-likelihood of all actions along successful trajectories while assigning zero weight to unsuccessful ones.

G. Scaling Overview

Table 3 summarizes the cumulative effect of each scaling dimension studied in §5.3.

H. Raw Experimental Results

We demonstrate the raw values of all experiments presented in §5 in Table 4 below.

Scaling	Legend in Figure 9	Peak (%)	Δ (%)
Baseline	Exclude Domains	31.0	–
+Breadth	Biased to Hard	34.5	+3.5
+Size	Only Easy	36.9	+2.4
+Depth	Uniform Sampling	38.2	+1.3
+Horizon Control	Shorten Horizon	42.9	+4.8

Table 3. **Scaling overview: peak performance of different scaling directions through RL settings on the test set.** Δ from Previous shows the improvement over the immediately preceding setting.

Method	H_{train}	Sampling	Domain	Memory	Rep. Penalty	Zero-shot (%)	RL Peak (%)
Instruct-8B	15 : 30 : 45	2 : 5 : 3	Complete	✓	✓	26.2	34.5±0.4
Thinking-8B	15 : 30 : 45	2 : 5 : 3	Complete	✓	✓	28.2	–
W/o Memory Prompt	15 : 30 : 45	2 : 5 : 3	Complete	✗	✓	27.4	31.5
W/o Repetition Penalty	15 : 30 : 45	2 : 5 : 3	Complete	✓	✗	26.2	33.0
Uniform Sampling	15 : 30 : 45	25 : 5 : 1	Complete	✓	✓	26.2	38.2
Only Easy	15 : 30 : 45	1 : 0 : 0	Complete	✓	✓	26.2	36.9
Only Medium	15 : 30 : 45	0 : 1 : 0	Complete	✓	✓	26.2	35.0
Exclude Domains	15 : 30 : 45	2 : 5 : 3	Half	✓	✓	26.2	31.0
Shorten Horizon	10 : 20 : 30	25 : 5 : 1	Complete	✓	✓	26.2	**42.9**
GPT-4o-SoM	–	–	–	✓	–	27.1	–
GPT-5-SoM (Think)	–	–	–	✓	–	29.8	–

Table 4. **Raw ablations and performance results of our trained agents on the OOD test set.** H_{train}: training horizon (easy:medium:hard steps); Sampling: difficulty ratio (easy:medium:hard); Domain: subdomain coverage; Memory: memory prompt; Rep. Penalty: repetition penalty. For Instruct-8B (Biased to Hard), we report the peak value of the averaged curve from two runs, with ± indicating the half-range.

I. Training Environment Designs

I.1. Rollout System: Operation-specific Local Request Queue

To avoid flooding the server with requests, the most naive approach is to implement a global queue (first-in-first-out rule) to avoid flooding the server with requests. However, below we show the ineffectiveness of this implementation because (1) it causes extensive warmup time and (2) it creates operation bottlenecks. To improve it, we demonstrate that each operation type should have its own queue. This is to say, requests with different types of operations should not have any priority over each other.

The computational model of rolling out a trajectory in a multi-step RL setting is shown in Figure 10 (*Left*). When simulation is required for an RL environment, there will be a sequence of CPU operations bounded by CPU resources. On the other hand, high-speed RL inference usually put high pressure on GPU resources as well. For the web agents rollout collection, the computational model can be simplified to five stages: navigating to a website with the simulator (N), taking a screenshot with the simulator (S), proposing an action with the agent, executing an action with the simulator, and sending the trajectory to an evaluator to obtain reward (E), with their speed defined on the right side of each box.

Assume we send a burst batch of 180 requests to the rollout system at $t = 0s$. As shown in Figure 10 (middle), with a global queue, at $t = 1s$, although 60 Navigation requests are finished, the Screenshot requests arrive *strictly after* the rest of the Navigation requests, so they have to wait for all Navigation requests to return. This makes GPU **starve** at $t = 2s$. Similarly, after $t = 5s$, only after all Screenshot requests are returned can the Execution requests be processed, and after

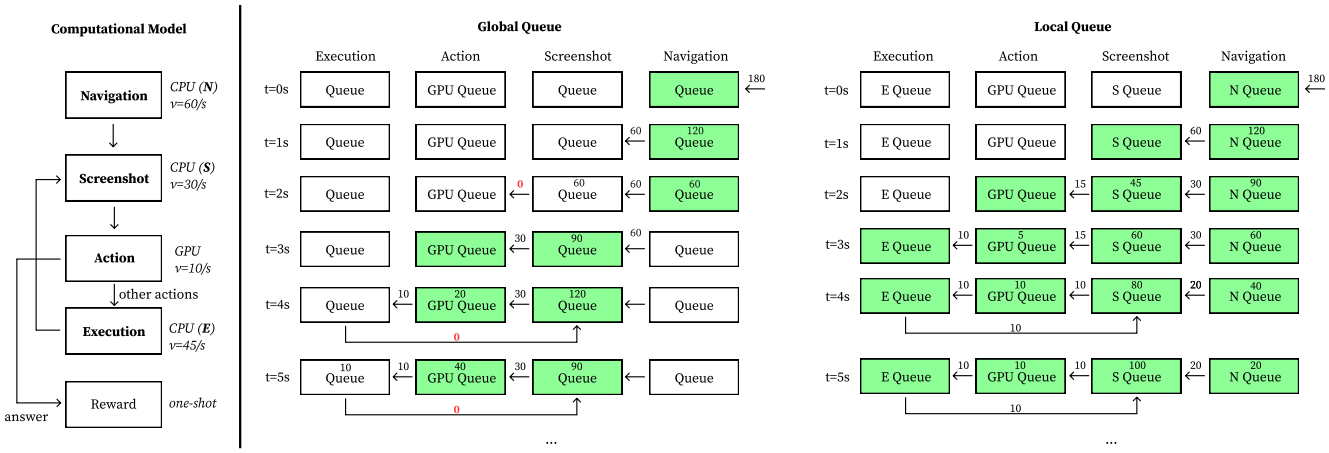

Figure 10. *WebGym* implements an ***operation-specific queue system*** that balances the CPU and GPU machine usages. Here we illustrate the computational model of this pool system in the *left* subplot, and the comparison of the two queue designs on the *right*. We represent the centralized CPU queue with *Queue*, and the cascading queue system for with the initial letter of the request type, specifically, Navigating to a webpage (**N**), taking a Screenshot (**S**), and Executing Action (**E**) that takes medium amount of time. A green box means the CPU machine is under optimal load.

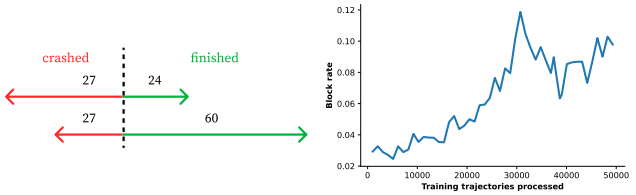

Figure 11. Left: Number of trajectories successfully finished and crashed out under high CPU load when $t = 10$min with 64 CPUs, 256 environments, and 3 GPU nodes. Top line: *global* queue. Bottom line: *local* queue. Right: **The block rate increases as training goes on.**

all Execution requests can be processed can the next-step Screenshot requests be processed, causing the GPU to **starve again and again** during the multi-step loop.

To improve this, *WebGym* implements a local queue that does not specify any priority *among* operations. At $t = 1s$, Screenshot requests and Navigation requests will be processed with the same priority. This results in 30 Navigation requests and 15 Screenshot requests being returned at $t = 2s$ (as they share CPU resources). When propagating this pattern through the pipeline, the GPU resources will not starve at any moment.

We experiment with the performance boost of this local queue under extreme CPU condition (64 CPUs needs to run 256 environments). Results are shown in Figure 11 (left). We observe that the crash percent decreases significantly if we employ the local queue, showcasing the effectiveness of this design.

I.2. Rollout Sampler: Anti-blocking

As we observe that some websites block frequent requests from the same server as shown in Figure 11 (right), we maintain a list of websites that blocks the agent with an LLM. The blocking is detected by GPT-4o with the prompt specified in §L.4.

J. Modeling the Markov Decision Process On Long-Horizon Web Navigation Tasks

A partial observable Markov decision process (POMDP) extends a Markov decision process (MDP) to settings where the agent does not have direct access to the underlying state but instead receives incomplete observations. Formally, a POMDP is defined as a tuple $(S, A, P, R, O, \Omega, \gamma)$, where S is the set of latent states, A is the action space, $P(s' \mid s, a)$ defines the state transition dynamics, $R(s, a)$ is the reward function, O is the observation space, $\Omega(o \mid s)$ specifies the observation model that maps hidden states to observations, and $\gamma \in [0, 1)$ is the discount factor. Because the agent does not directly observe s_t, it maintains a belief state b_t, a probability distribution over S, which it updates using past actions and observations. As a result, the optimal policy maps belief states to actions, $\pi(b_t) \to a_t$, instead of mapping fully observed states to actions as in standard MDPs.

Recent work models the web agent tasks as a POMDP, because historical observations must be included to successfully infer the next action. Formally, for a web agent task, if we represent the observation (including the screenshot and metadata of that webpage) at step t as o_t, recent work represents the true state s_t with $s_t = o_{t-n:t}$ (concatenation of the observations of last n steps) [5, 34]. This is called a **windowed history**, designed to fit into the context limit of the vision-language models. Formally, this modeling assumes the approximation

$$\Pr\big(o_{t+1} | o_{\mathbf{1}:t}, a_{\mathbf{1}:t-1}, c\big) \approx \Pr\big(o_{t+1} | o_{\mathbf{t-n}:t}, a_{\mathbf{t-n}:t-1}, c\big), \tag{2}$$

where o is the vector that represents observation, and c is the vector that represents task [5, 31, 34, 46]. In words, these works treat observations of the last several steps as sufficient statistics of history for predicting future observations. In

practice, the window size n is usually set to 3. For long-horizon tasks, this modeling over-simplifies the problem because the policy will frequently return to the same state. If the policy is not aware of its full history, it can produce repetitive sequences of actions because it is not aware of what it already finished earlier.

Previously, there have been clever designs in token-level RL (modeling a token as a step), e.g. RL from human feedback (RLHF), that mitigate this problem by directly represent s_t with the representation of the token from the auto-regressive vision-language model, which automatically encodes information from all previous steps, naturally enforcing $s_t = o_{1:t}$. Is there a way we can bring this into sequence-level RL (modeling a sequence as a step)?

Practically, the problem with inputting full history observations $o_{1:t}$ to the VLM policy is that the context window of even the most advanced VLMs is not large enough, partially because image is too dense a representation for web agents tasks. Recent work like GLM-4.1v [16] addresses this limitation by introducing a **memory** as part of the tokens that the VLM should output in each step. Formally, the memory at step t, denoted by m_t, is a result of the observation without memory $\tilde{o}_t$, the memory from the previous step m_{t-1}, and the task instruction c, denoted as $m_t = \text{VLM}_\theta(\tilde{o}_t, m_{t-1}, c)$, and the observation at step t can now be denoted as $o_t = (m_t, \tilde{o}_t, c)$. This design leads to a continuous compression of the memory based on the progress of the task, which resembles RLHF and Long-Short-Term Memory (LSTM). We practice this idea for our agent context management, as shown in Section L.1.

K. Qualitative Examples

K.1. Example Rubrics

We show some example rubrics generated in Table 5. We can observe a general correlation between the instruction length of the task and the number of criteria in its rubrics.

Table 5. Examples of WebGym Tasks with Evaluation Rubrics

Task	**Evaluation Rubric**
Find out the date of the next Baden Marathon event.	**Group 1:** Find date of next Baden Marathon event *Facts:* - date of the next Baden Marathon event
Find the recommended daily feeding amount for an adult British Shorthair cat (over 12 months) on royalcanin.pl, assuming a target weight of 5kg and a moderate activity level.	**Group 1:** Identify the target cat specifications *Facts:* - adult British Shorthair cat - age over 12 months - target weight of 5kg - moderate activity level **Group 2:** Find recommended daily feeding amount *Facts:* - recommended daily feeding amount for the specified cat on royalcanin.pl
Find the highest-rated mover in Honolulu to shift a vehicle and large appliances and who has virtual discussion options available in Yelp.	**Group 1:** Identify highest-rated mover in Honolulu *Facts:* - mover is located in Honolulu - mover is highest-rated **Group 2:** Services provided by the mover *Facts:* - mover offers vehicle shifting services - mover offers large appliances shifting services **Group 3:** Communication options available *Facts:* - mover has virtual discussion options available **Group 4:** Source of information *Facts:* - information is found on Yelp

K.2. Comparing Rubric-based Evaluation with Task-based Evaluation

We find many cases where rubric-based evaluation is able to produce significantly more comprehensive evaluation that checks details of each criteria of the task, which leads to many wins over using only the task description. One example is shown in Figure 12. The human evaluation for this example is NOT SUCCESS.

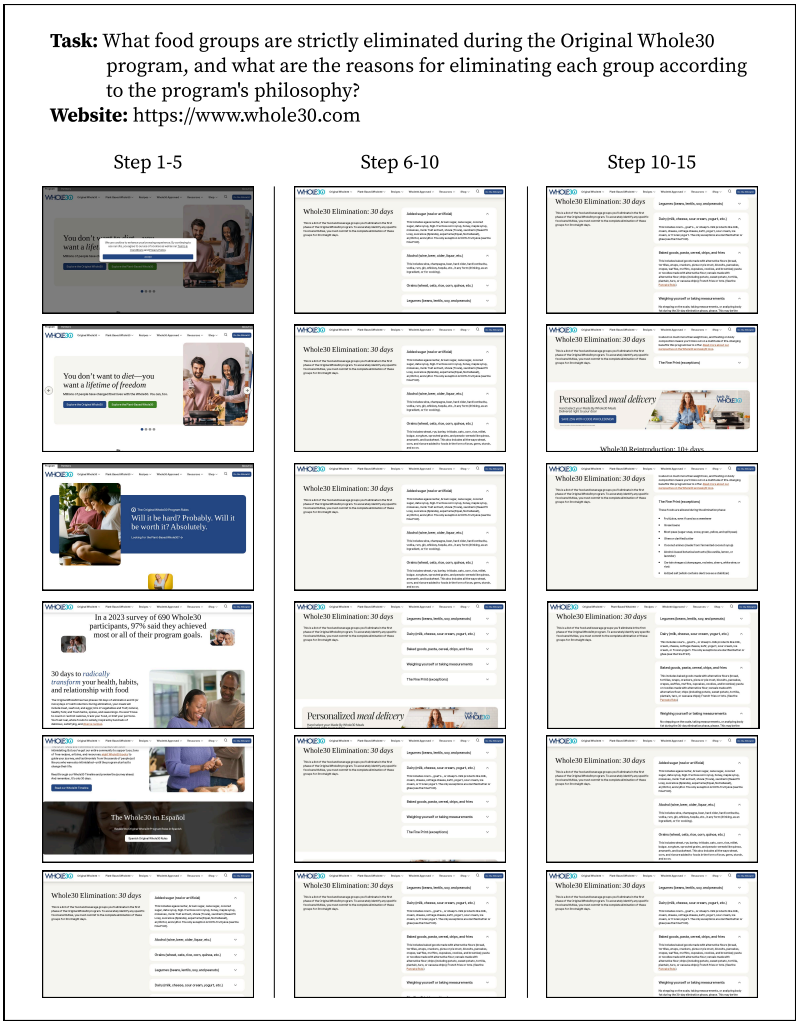

Figure 12. Trajectory for the comparison between vanilla evaluation and rubric-based evaluation.

Task description-based evaluation.

- **Verify criterion a:** The response mentions the food groups eliminated during the Original Whole30 program: added sugar, alcohol, grains, legumes, dairy, and baked goods, pasta, cereal, chips, and fries. The screenshots provide evidence of these eliminations as listed on the Whole30 website. Additionally, the reasons mentioned in the response for eliminating these groups, such as identifying food sensitivities and improving relationship with food, are also corroborated by the screenshots, particularly those discussing the philosophy of Whole30.
- **Verify criterion b:** The task asked for the food groups strictly eliminated during the Original Whole30 program and the reasons for their elimination according to the program's philosophy. The agent's response matches the task requirements accurately, listing the correct food groups and providing the reasons as asked, based on the information available in the screenshots.
- **Verdict: SUCCESS**

Rubric-based evaluation.

- There are 2 groups for this rubric in total.
- **Group 1: Identify food groups eliminated during the Original Whole30 program**
- – **Description:** Identify food groups eliminated during the Original Whole30 program
  – **Facts:** Identify the food groups strictly eliminated during the Original Whole30 program
  – **Verify criterion a:** The response claims that the food groups strictly eliminated during the Original Whole30 program include added sugar, alcohol, grains, legumes, dairy, and items such as baked goods, pasta, cereal, chips, and fries. The screenshots clearly show a list of eliminated food groups that includes added sugar (real or artificial) alongside the other listed categories. Therefore, the response's identification of eliminated food groups is supported by the screenshots.

  - **Verify criterion b:** This group requires identifying the food groups strictly eliminated during the Original Whole30 program. The response provides a concrete list of eliminated groups (including added sugar), which directly satisfies the identification requirement.
  - **Verdict:** SUCCESS
- **Group 2: Reasons for eliminating each food group**
- – **Description:** Reasons for eliminating each food group
  - **Facts:** Reasons for eliminating each food group according to the Original Whole30 program's philosophy
  - **Verify criterion a:** While the screenshots list the eliminated food groups, they do not explicitly provide the program's philosophy-based reasons for eliminating each group. The response offers general motivations (e.g., identifying sensitivities or resetting habits), but these statements are not directly verifiable from the screenshots as provided.
  - **Verify criterion b:** This group specifically asks for reasons for eliminating *each* food group according to the Original Whole30 philosophy. The response does not provide distinct, food-group-specific reasons (it stays general rather than explaining each eliminated group), and the reasons it does provide are not grounded in the screenshots. Therefore, the response does not meet the requirement of explaining why each food group is eliminated.
  - **Verdict:** NOT SUCCESS
- **Final verdict: NOT SUCCESS**

L. Prompts

L.1. Agent Action Prompt

The prompt we use to rollout trajectories is the agent action prompts shown below. Both the official prompt as provided in QwenLM-Team [31] and the memory prompt that we use across §5 are included below.

Prompt: Agent Action Generation

```
================================================================================
MESSAGE 1: SYSTEM
================================================================================
You are a helpful assistant.

# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools> XML tags:
<tools>
{
 "name": "computer_use",
 "description": "Use a mouse and keyboard to interact with a computer. The screen's resolution is 1000x1000.
* You do not have access to download files or play videos.
* Focus on web browsing and navigation tasks only.",
 "parameters": {
  "type": "object",
  "properties": {
   "action": {
    "type": "string",
    "description": "The action to perform:
* 'left_click': Click the left mouse button at the specified coordinates.
* 'type': Type text at the specified coordinates. The system will automatically click at the coordinates, type the text, and press Enter.
* 'scroll': Scroll the page in the specified direction (up or down).
* 'wait': Wait for the specified number of seconds for changes to occur.
* 'go_back': Go back to the previous page in browser history.
* 'navigate': Navigate directly to a specific website URL. The URL must start with https://. CRITICAL: If you see reCAPTCHA or any
    CAPTCHA challenge on the screen, DO NOT attempt to solve it. Instead, immediately use the navigate action to go to a different relevant
    website to complete your task.
* 'answer': Provide the final answer to complete the task.",
    "enum": ["left_click", "type", "scroll", "wait", "go_back", "navigate", "answer"]
   },
   "coordinate": {
    "type": "array",
```

”description”: ”[x, y] coordinates (0−1000 range). Required for left_click and type actions. For type action, specify WHERE to type (e.g., coordinates of input field).”,
”items”: {
”type”: ”integer”,
”minimum”: 0,
”maximum”: 1000
},
”minItems”: 2,
”maxItems”: 2
},
”text”: {
”type”: ”string”,
”description”: ”Text to type or answer. Required for type and answer actions. Note: For type action, the system will automatically click at the coordinates, type the text, and press Enter − no need to click separately before typing.”
},
”direction”: {
”type”: ”string”,
”enum”: [”up”, ”down”],
”description”: ”Scroll direction. Required for scroll action.”
},
”time”: {
”type”: ”number”,
”description”: ”Seconds to wait. Required for wait action.”
},
”url”: {
”type”: ”string”,
”description”: ”URL to navigate to. Required for navigate action. Must start with https://.”
}
},
”required”: [”action”]
}
}
</tools>

For each function call, return a JSON object with function name and arguments within <tool_call></tool_call> XML tags:
<tool_call>
{”name”: <function−name>, ”arguments”: <args−json−object>}
</tool_call>

Response format

{{if w/ Memory Prompt}}
Response format for every step:
1) Memory: facts you would like to memorize for future actions in json format. Include the current step.
2) Progress: Decompose the task into subtasks and what has been finished so far with json format. Include progress of the current step.
3) Intention: clearly state which subtask you’re working on at this step with the json key.
4) Action: a short sentence describing what to do in the UI to accomplish the next subtask.
5) A single <tool_call>...</tool_call> block containing only the JSON: {”name”: <function−name>, ”arguments”: <args−json−object>}.

Rules:
− Output exactly in the order: Memory, Progress, Intention, Action, <tool_call>.
− You MUST use json format for the Memory and Progress parts.
− Example Task: ”Search and compare the prices and locations of product 1 and product 2 on Amazon.”
  − Example of Memory json format: {”Price of product 1”: ”10.00”, ”Location of product 1”: ”10.00”, ”Price of produce 2”: ”12.00”}.
  − Example of Progress json format: {”Go to Amazon.com”: ”finished”, ”Search for price of product 1”: ”finished”, ”Search for location of product 1”: ”finished”, ”Search for price of product 2”: ”finished”, ”Search for location of product 2”: ”not finished”, ”Compare product 1 and product 2”: ”not finished”}.
  − Example of Intention json key format: ”Search for location of product 2”.
− You CAN NOT modify previous Memory. Only append to it.
− You CAN modify Progress from previous conversation to further decompose the task and guide your next action.
  − For example, if the previous assistant message specifies Progress: {”Go to Amazon.com”: ”finished”, ”Search for product 1”: ”finished”, ”Search for product 2”: ”not finished”, ”Compare product 1 and 2”: ”not finished”},
  − You should further decompose ”Search for product 1” and ”Search for product 2” into ”Search for price of product 1” and ”Search for location of product 1”, and ”Search for price of product 2” and ”Search for location of product 2”.
− Do not output anything else outside those five parts.”””

{{/if w/ Memory Prompt}}

{{if w/o Memory Prompt}}

Response format for every step:
1) Action: a short sentence describing what to do in the UI.
2) A single <tool_call>...</tool_call> block containing only the JSON: {"name": <function−name>, "arguments": <args−json−object>}.

Rules:
− Output exactly in the order: Action, <tool_call>.
− Action describes the high−level intention of the tool call within a single sentence.
− Do not output anything else outside those two parts.

{{/if w/o Memory Prompt}}

==========
MESSAGE 2: USER
==========
[IMAGE: Screenshot of current webpage]
Please generate the next action according to the UI screenshot and task.

Task: Find the details of the public transport journey from Glasgow Queen Street to Edinburgh Waverley, departing on May 22, 2025, at 09:00 am.

Initial website: https://www.nationalrail.co.uk

Generate the next action to complete the task.
==========
MESSAGE 3: ASSISTANT {{Example response, note that when using the w/ memory prompt, the previous assistant message will contain memory response}}
==========
Action: Click on the search input field to enter the departure station.

<tool_call>
{"name": "computer_use", "arguments": {"action": "left_click", "coordinate": [245, 312]}}
</tool_call>

==========
MESSAGE 4: USER
==========
[IMAGE: Screenshot after clicking]

==========
MESSAGE 5: ASSISTANT {{Example response, note that when using the w/ memory prompt, the previous assistant message will contain memory response}}
==========
Action: Type the departure station name into the search field.

<tool_call>
{"name": "computer_use", "arguments": {"action": "type", "coordinate": [245, 312], "text": "Glasgow Queen Street"}}
</tool_call>

L.2. Evaluator Prompt: Keypoint Detection

Prompt: Evaluator/Evaluation Criteria Generation

You are an expert evaluator determining whether an image contains relevant information for completing a task.

∗∗Instructions∗∗:
− Answer "YES" if the image shows ANY task−related content: actions taken, progress, search results, tool usage, error messages, or blocking screens.
− Answer "NO" only if completely irrelevant: generic homepage, unrelated webpage, or blank screens with no context.
− When in doubt, answer "YES".

∗∗Response format∗∗:

1. **Reasoning**: [One sentence explanation]
2. **Decision**: [YES or NO]

User Prompt:

Task: {task}
Key Points for Task Completion:
{eval_rubric}

The snapshot of the web page is shown in the image. Does this image contain relevant information for the task? (Answer YES unless it's completely irrelevant)

Where:
– {task} = task.task_name (the task description)
– {eval_rubric} = List of all criteria from task.evaluator_reference

Example filled in:
Task: Find the price of iPhone 15 Pro on Amazon
Key Points for Task Completion:
– The agent should provide the exact price of the iPhone 15 Pro
– The price must be visible in the screenshot
– The product must be from Amazon

L.3. Evaluator Prompt: Per-criteria Evaluation

The evaluation system uses a two-criterion approach, each checked separately:

- Criterion A is checked per-fact (each rubric/fact from task decomposition is evaluated separately).
- Criterion B is checked once per task (verifies the agent's final answer against screenshots).

L.3.1. Criterion A: Fact Verification

Prompt: Evaluator/Criterion A - Fact Verification

==========
MESSAGE 1: SYSTEM
==========
You are an expert in evaluating the performance of a web navigation agent. The agent is designed to help a human user navigate a website to complete a task. Your goal is to verify whether a SPECIFIC FACT can be confirmed by the provided screenshots.

As an evaluator, you will be presented with the following components:

1. Task Instruction: The original task description (provided for CONTEXT ONLY)
2. Fact Group: A group of related facts decomposed from the task instruction (provided for CONTEXT ONLY)
3. Fact to Check: A specific fact that you need to verify (THIS IS YOUR PRIMARY FOCUS)
4. Trajectory: A complete list of observations and actions that were taken by the agent
5. Result Screenshots: Visual representation of the screen showing the result or intermediate state

CRITICAL: Your judgment should ONLY focus on whether the FACT TO CHECK can be verified by the screenshots. You are NOT checking the agent's response – only whether the screenshots contain evidence for the fact.

Guidelines for evaluation:
−− Your primary responsibility is to assess whether the screenshots contain evidence that verifies the FACT TO CHECK.
−− The fact to check may involve more than one sub−fact. ALL sub−facts must be verifiable from the screenshots.
−− If the fact requires specific information (e.g., "concert is in the US or Canada"), the screenshots must show this information.
−− If the evaluation criteria asks to find a specific item, the screenshots must show that exact item (not a similar one).

IMPORTANT – Handling "OR" conditions:
−− When the fact or task contains "OR" (e.g., "best books on cooking OR gardening OR home decor"), satisfying ANY ONE of the alternatives is sufficient for SUCCESS.
−− Example: If the task is "find best books on cooking OR gardening OR home decor" and the screenshots show best cooking books, this is SUCCESS – the agent does NOT need to find all three.
−− "OR" indicates alternatives/options, not a requirement to verify all items.

Response format (you should STRICTLY follow the format):
1. Analysis: [Describe what evidence you see in the screenshots related to the fact to check]

2. Verdict: [SUCCESS if the fact is verified by screenshots, NOT SUCCESS otherwise]

==
MESSAGE 2: USER
==
===Your Turn===
Task Instruction (for context only):
[task_instruction]

Fact Group (for context only):
[fact_group]

Fact to Check (PRIMARY FOCUS – verify this against the screenshots):
[fact_to_check]

Completion history:
[trajectory]

Relevant screenshots:
attached.

Evaluation: (MUST end with line "2. Verdict: [SUCCESS or NOT SUCCESS]")

L.3.2. Criterion B: Response Verification / Anti-Hallucination

Prompt: Evaluator/Criterion B - Anti-Hallucination Check

==
MESSAGE 1: SYSTEM
==
You are an expert in detecting hallucinations in web navigation agent responses. Your goal is to verify whether the agent's FINAL RESPONSE is supported by the provided screenshots.

As an evaluator, you will be presented with the following components:

1. Task Instruction: The task the agent was trying to complete
2. Final Response: The agent's answer/response to the task
3. Result Screenshots: Visual representation of the screens the agent visited

CRITICAL: Your job is to check if the agent's response contains information that is NOT shown in the screenshots. Agents frequently hallucinate or make up answers that are not verified by what they actually saw.

Guidelines for evaluation:
– Check whether EVERY claim in the agent's response can be verified by the screenshots.
– If the response mentions specific facts (names, numbers, dates, locations, etc.), these MUST be visible in the screenshots.
– If the response contains information not shown in ANY screenshot, this is a hallucination – mark as NOT SUCCESS.
– If the response is vague or says "I couldn't find the information", check if this matches what the screenshots show.
– YOU SHOULD EXPECT THAT THERE IS A HIGH CHANCE THAT THE AGENT WILL MAKE UP AN ANSWER NOT VERIFIED BY THE SCREENSHOTS.

Response format (you should STRICTLY follow the format):
1. Claims in response: [List the specific claims/facts in the agent's response]
2. Screenshot verification: [For each claim, state whether it appears in the screenshots]
3. Verdict: [SUCCESS if ALL claims are verified by screenshots, NOT SUCCESS if ANY claim is not verified]

==
MESSAGE 2: USER
==
===Your Turn===
Task Instruction:
[task_instruction]

Agent's Final Response:
[response]

Screenshots from the agent's session:
attached.

Evaluation: (MUST end with line "3. Verdict: [SUCCESS or NOT SUCCESS]")

L.4. Evaluator Prompt: Blocking Detection

Additionaly, we introduce a blocking detection step, which is used to identify if failures were due to anti-bot measures vs. agent errors. Those websites that frequently blocks the agent will be excluded from being sampled.

Prompt: Evaluator/Blocking Detection

==========
MESSAGE 1: SYSTEM
==========
You are an expert in analyzing web navigation trajectories to detect if a website has EXPLICITLY and UNRECOVERABLY BLOCKED an automated agent using anti−bot measures.

CRITICAL DISTINCTION:
− Blocked = Website explicitly prevents access with UNRECOVERABLE anti−bot measures that the agent CANNOT escape from by navigating
− NOT Blocked = Agent makes navigation mistakes, clicks wrong elements, or encounters temporary obstacles that could be navigated away from

EXPLICIT SIGNS OF UNRECOVERABLE BLOCKING (answer YES):
− CAPTCHA challenges that persist and cannot be bypassed (reCAPTCHA, hCaptcha, image selection puzzles)
− Cloudflare protection screens that remain stuck ("Checking your browser", "Just a moment", "DDoS protection")
− Explicit "Access Denied" or "403 Forbidden" error pages that appear consistently
− "Please verify you are human" prompts that block all access
− Bot detection pages that explicitly state "automated access detected" and prevent further navigation
− Security challenge pages that completely prevent any website access

DO NOT COUNT AS BLOCKING (answer NO):
− Agent clicked wrong buttons or links (navigation mistakes)
− Agent cannot find information on a normally accessible page (task difficulty)
− Page loaded slowly but eventually loaded (performance issue, not blocking)
− Agent got lost or confused navigating the site (agent error)
− "Page not found" or 404 errors (broken links, not blocking)
− Empty search results (legitimate website response, not blocking)
− Temporary errors that the agent could escape from by going back or navigating elsewhere

CRUCIAL: Only answer "YES" if you see PERSISTENT, UNRECOVERABLE anti−bot measures that prevent ALL navigation on the website.

Response format (you MUST strictly follow):
1. Analysis: [Describe what you see in the screenshots − any EXPLICIT signs of anti−bot blocking measures?]
2. Blocked: [YES or NO]

==========
MESSAGE 2: USER
==========
===Your Turn===
Task:
[task]

Trajectory:
[trajectory]

Screenshots:
attached.

Did the website block the agent? (MUST end with line "2. Blocked: [YES or NO]")

L.5. Task Set Generation Prompt: Evaluation Criteria Proposal

Prompt: Task/Evaluation Criteria Generation

======== Guidelines ========
You're a helpful assistant that generates fact−based evaluation rubrics for web navigation tasks.

Your task is to:
1. Break down the task into hierarchical fact groups, where each group contains specific facts to verify
2. Each fact represents ONE verifiable piece of information that can be checked in a trajectory
3. The overall task difficulty equals the TOTAL NUMBER of all facts across all groups

Key principles:
− FIRST analyze the task to identify logical groupings of related information
− For each group, list ALL specific facts that need to be verified
− Facts should be AS DETAILED AS POSSIBLE − break down complex requirements into individual checkable facts
− Each fact has a difficulty of 1 (facts are atomic units)
− The OVERALL DIFFICULTY is the TOTAL COUNT of all facts across all groups
− A trajectory is marked correct if and only if ALL facts are verified
− Do not make up facts not present in the original task

======== CRITICAL: Do Not Decompose Mathematical Problems ========

When a task involves mathematical calculations, physics problems, or computational work, DO NOT break down the calculation steps. Treat the entire computation as a SINGLE fact.

BAD Example:
Task: What is the result of 2^3 + 2^2 + 2^1 + 2^0?

{
 "fact_groups": [{"id": 1, "facts": ["calculate 2^3", "calculate 2^2", ...]}],
 "difficulty": 5
}
Problem: Breaking down calculation into individual steps.

CORRECT:
{
 "fact_groups": [{"id": 1, "facts": ["result of the equation 2^3 + 2^2 + 2^1 + 2^0"]}],
 "difficulty": 1
}

======== CRITICAL: Do Not Decompose "How−To" Instructions ========

When a task asks "How to" do something, the task is asking for INSTRUCTIONS from the website, not to verify each step was performed.

BAD Example:
Task: How can I set up an Apple Watch for the first time?

{
 "fact_groups": [
   {"id": 1, "facts": ["ensure Apple Watch is charged", "check compatibility"]},
   {"id": 2, "facts": ["turn on Apple Watch", "bring close to iPhone", ...]}
 ],
 "difficulty": 11
}
Problem: Breaking down procedure into action steps.

CORRECT:
{
 "fact_groups": [{"id": 1, "facts": ["instructions for setting up Apple Watch for the first time"]}],
 "difficulty": 1
}

======== CRITICAL: AND vs OR Logic ========

ALL facts must be verified − facts use AND logic by default.

If the task contains OR statements (e.g., ”do X or Y”), represent as a SINGLE fact with OR capitalized.

BAD Example:
Task: What should a user do if they were trying to contact the original owners or access the previous content.

{
 ”fact_groups”: [
  {”id”: 1, ”facts”: [”method to contact the original owners”]},
  {”id”: 2, ”facts”: [”method to access the previous website content”]}
 ]
}
Problem: Treating OR as AND (creating separate groups).

CORRECT:
{
 ”fact_groups”: [
  {”id”: 1, ”facts”: [”method to contact the original owners OR access the previous website content”]}
 ],
 ”difficulty”: 1
}

======== CRITICAL: Do Not Enumerate Unspecified Items ========

When a task asks to ”identify” or ”find” items WITHOUT specifying what they are, DO NOT make up specific facts. Similarly, do not enumerate ”details” or ”information” into specific types.

BAD Example:
Task: What are the contact details for Hamble Harbour Office?

{
 ”fact_groups”: [{”id”: 1, ”facts”: [”telephone number”, ”email address”, ”physical address”]}],
 ”difficulty”: 3
}
Problem: Enumerating specific types when task just says ”contact details”.

CORRECT:
{
 ”fact_groups”: [{”id”: 1, ”facts”: [”contact details for Hamble Harbour Office”]}],
 ”difficulty”: 1
}

======== CRITICAL: ”Such as” Are Illustrative, Not Facts ========

Phrases like ”such as”, ”for example”, ”e.g.” provide examples to clarify, NOT additional facts.

BAD Example:
Task: Find a vegetarian lunch dish, such as chickpea salad or lentil soup, with 5−star rating.

{
 ”fact_groups”: [{”id”: 1, ”facts”: [”recipe is vegetarian”, ”example dishes include chickpea salad or lentil soup”]}]
}
Problem: Creating fact for examples.

CORRECT:
{
 ”fact_groups”: [{”id”: 1, ”facts”: [”recipe is for a vegetarian dish”, ”dish is suitable for lunch”, ”recipe has a 5−star rating”]}],
 ”difficulty”: 3
}

======== Example ========

Task: Find the full list of exhibitions at The Ringling, including their titles, dates, and descriptions.

Expected response:
{
 ”fact_groups”: [

```
    {
     "id": 1,
     "description": "find complete list of exhibitions at The Ringling",
     "facts": [
       "full list of exhibitions",
       "current exhibitions included",
       "upcoming exhibitions included",
       "exhibitions are at The Ringling"
     ]
    },
    {
     "id": 2,
     "description": "get details for each exhibition",
     "facts": [
       "title for each exhibition",
       "specific dates for each exhibition (if available)",
       "brief description for each exhibition"
     ]
    }
  ],
  "difficulty": 7
}

======== Output Format ========
Respond with valid JSON:
- "fact_groups": Array of objects with "id", "description", "facts" (array of strings)
- "difficulty": Integer = TOTAL COUNT of all facts across all groups

======== Your Turn ========
Task: {task}
>>>>>>>>>>>
```

L.6. Task Set Generation Prompt: Task Proposal from Fact Groups

Prompt: Task/Task Generation from Selected Fact Groups

```
======== Task Generation ========

You are a helpful assistant that generates simplified task descriptions based on selected fact groups from the original task.

Original Task: {original_task}

Selected Fact Groups:
{selected_groups}

Your task is to:
1. Create a new task description that incorporates ONLY the requirements explicitly listed in the selected fact groups above
2. The new task should read naturally and be self-contained
3. Do NOT include requirements from fact groups that were not selected
4. Maintain the original context and domain of the task
5. **CRITICAL**: Do NOT add any information that is not explicitly present in the selected fact groups above, even if that information appears in the original task

Important Guidelines:
- If a fact group mentions "concert" but does NOT mention "upcoming" or "in the US or Canada", do NOT include those constraints in the generated task
- ONLY use constraints and requirements that are explicitly listed as facts in the selected groups
- Do not infer or add information from the original task that is not in the selected fact groups

Output Format:
Respond with ONLY the new task description as plain text (no JSON, no extra formatting).

Example:
```

Original Task: ”Find the full list of current and upcoming exhibitions at The Ringling, including their titles, specific dates (if available), and a brief description for each.”

Selected Fact Groups:
– Group 1 (find complete list of exhibitions at The Ringling): full list of exhibitions, current exhibitions included, upcoming exhibitions included, exhibitions are at The Ringling

New Task: ”Find the full list of current and upcoming exhibitions at The Ringling.”

Your Turn:
>>>>>>>>>>>>

M. Hyper-parameters

Our system consists of four main components: environment management, data collection (rollout), model training, and the agent model. The hyperparameters for each component are carefully tuned to balance performance, scalability, and sample efficiency. The final hyperparameters can be found in Table 6.

Table 6. **Hyperparameters for All Experiments**. Bolded option suggests an experimented optimality.

Component	Hyperparameter	Value
Environment (2 workers)	Simulation CPU Server size	80
	HTTP pool size (screenshot/execute)	256
	HTTP pool size (metadata/navigate)	256
	HTTP pool size (allocate/release)	4
	Max vLLM sessions	80
Rollout	Train tasks rollout size	512, **1024**
	Task Sampling	**uniform**/ratio
	Max steps (easy,medium,hard)	(15,30,45), **(10,20,30)**
	Interaction mode	**coordinates-only**, set-of-marks
	Temperature	1.0
	Top-p	0.99
	Top-k	2
Training	Learning rate	1e-5, 5e-6, **1e-6**
	Effective batch size	3*4*8*2
	Times each sample is trained	≈ 2
	Max gradient norm	1.0
	Weight decay	0, **0.01**
	Warmup steps	**30**, 100
	LR scheduler type	**constant_with_warmup**, cosine
	Samples to train per iteration	1200, **1800**
	Recency bias power	1, **2**
	Replay buffer capacity	**last 4**, last 8
	Cutoff length	4096, 8192, **16384**
	Gradient checkpointing	False
	bf16 precision	True
Model	Model type	Qwen3-VL-8B
	Variant	**Instruct, Thinking**
	Max new tokens	1024, 2048, **3072**