ONE ROLLOUT IS ALL YOU GET: FULLY TEST-TIME ADAPTATION FOR GUI AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

GUI agents are deployed with frozen weights and discard everything they experience on the job. Existing ways to update an agent's weights assume something deployment withholds: ground truth, rollouts beyond the single attempt (retries, samples, practice runs), or a learning phase other than deployment. Because GUI actions can be irreversible, a deployed agent gets one attempt per task occurrence, in arrival order, and every attempt counts. No ground truth is available at any point. We define *fully test-time adaptation* for GUI agents by these constraints and pair it with a minimal weight-space method, SOLO. Auxiliary models read each episode: a judge selects the episodes it deems successful, and a proposer–verifier pair relabels a failed episode's prefix with the subtask that prefix completed. Admitted episodes enter a short sliding window, and each admission updates a small adapter by top-K self-distillation on the agent's own predictions, provided the window holds a judged success. On recurring task streams built from WebArena, VisualWebArena and MobileWorld, SOLO improves on the frozen agent with both UI-TARS-7B and Qwen3-VL-8B, by three to six points of success rate, and exceeds two in-setting memory methods on the web streams.

1 INTRODUCTION

A GUI agent that books flights, files issues, or manages a calendar on a user's behalf (Qin et al., 2025; Liu et al., 2025; Bai et al., 2025) ships with fixed weights, and from then on everything it experiences is discarded. Yet the deployment stream is the distribution the agent is judged on, and for many deployments that distribution recurs. A personal agent is the clearest case: a single user returns to the same apps and accounts, with the same errands coming back as new instances, the same expense form with a new amount, the same store with a new item. An agent that cannot learn from this stream makes the same mistake on the tenth occurrence of a task as on the first.

Most methods for improving an agent after it ships rely on a resource that deployment does not provide, and Tab. 1 places them by the resources they use: ground-truth supervision, rollouts beyond the single attempt that counts, and a stage other than deployment at which to adapt. Demonstration fine-tuning trains on labeled trajectories before deployment (Wu et al., 2025; Qin et al., 2025). Reinforcement learning and the self-evolution flywheels built on it learn from a reward, given by the environment or a trained reward model, with parallel rollouts, resets and an iterated training phase (Bai et al., 2024; Qi et al., 2025; Xiao et al., 2025). Exploration memory is filled by practice runs before the attempts that count (Zhang et al., 2025; Sun et al., 2026), and retry-based improvement attempts a task again (Shinn et al., 2023; Li et al., 2026a; He et al., 2026). Multi-rollout inference and adaptation draw many samples of each input to select among at inference or to train on at test time, or train on data derived from each test input (Snell et al., 2024; Yang et al., 2026; Zuo et al., 2025; Akyürek et al., 2025). A live deployment of a GUI agent provides none of these resources.

A GUI action can be irreversible: an email sent is sent, a purchase placed is placed. An agent working in a real account cannot fork the world to sample many candidate action sequences, replay a task to try another route, or reset the environment between attempts. It gets exactly one attempt per task occurrence, in the order the user issues tasks, and every attempt counts, because each is an errand the user wanted done. We also assume no ground truth: no checker inspects the account, no annotator labels the outcome, and the user gives no feedback. Whatever feedback exists must be computed from the episode itself by automated means, and unlike ground truth it can be wrong.

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.

Table 1: **Where fully test-time adaptation sits among ways to improve a GUI agent.** Each row is placed by the resources its learning uses: ground-truth supervision (demonstrations, an environment check or a human; *Varies*: some members use it, or a trained or VLM evaluator stands in), rollouts beyond the single attempt that counts, the stage at which adaptation happens, and the state retained across tasks. Our setting withholds the first two and admits only the deployment stage, with memory or weights as the retained state. The row marked † satisfies it. The last column lists representative works, and Sec. 5 discusses the rest.

Approach	GT supervision	Extra rollouts	Adaptation stage	Retained state	Related work
Demonstration SFT	Yes	No	Pre-deployment	Weights	Wu et al. (2025); Qin et al. (2025)
RL / self-evolution	Varies	Yes	Pre-deployment	Weights	Bai et al. (2024); Qi et al. (2025); Xiao et al. (2025)
Exploration memory	Varies	Yes	Pre-deployment	Memory	Zhang et al. (2025); Sun et al. (2026)
Retry-based improvement	Varies	Yes	Deployment	Memory	Shinn et al. (2023); Li et al. (2026a); He et al. (2026)
Multi-rollout inference	No	Yes	Deployment	None	Snell et al. (2024); Yang et al. (2026)
Multi-rollout adaptation	No	Yes	Deployment	Weights	Akyürek et al. (2025); Zuo et al. (2025)
Single-pass memory†	No	No	Deployment	Memory	Wang et al. (2024); Mi et al. (2026); Wang et al. (2025b)
SOLO (ours)	**No**	**No**	**Deployment**	**Weights**	

Following Tent (Wang et al., 2021), which defined fully test-time adaptation for perception models by what it withholds (the correspondence is spelled out at the end of Sec. 2), we define *fully test-time adaptation* for GUI agents by three constraints: no ground truth in any form, no rollouts beyond the single attempt per task occurrence (no retries, samples or exploration episodes), and no learning phase other than deployment, including practice runs before the tasks that count. Any signal that automated models compute from the agent's own episodes is permitted, and so is any state that persists across the stream, external memory or weights alike. What remains is an off-the-shelf agent, its tasks in arrival order, and one attempt per task occurrence. Single-pass memory systems driven by a model judge (Wang et al., 2024; Mi et al., 2026) already satisfy these constraints, and every other approach in Tab. 1 relies on at least one withheld resource.

The first design choice of our method, SOLO, is the learning signal. Tent trusts the model's confidence and minimizes its entropy, but for a GUI agent entropy minimization leaves the success rate at the frozen level on WebArena and collapses the agent on VisualWebArena (App. A). SOLO therefore lets auxiliary models read each episode. A judge decides whether the episode completed its task (Pan et al., 2024) and thereby selects what to learn from. For a judged failure, a proposer names a subtask that a prefix of the episode completed, as in hindsight relabeling (Andrychowicz et al., 2017; Zhang et al., 2023), and an independent verifier checks the claim against the screenshots before the prefix is admitted. None of these models generates an action or a token target: they decide which of the agent's own episodes to learn from and under which instruction.

A judged success enters a short sliding window of recent admissions, and an admitted relabeled prefix enters the same window in place of the failed episode. Each admission triggers one small update of a low-rank adapter on the frozen backbone by top-K self-distillation: the targets are the policy's own top-K predictions at every position, so the executed tokens are never imitated outright. Updates fire only while the window holds at least one judged success, so a relabeled prefix is trained only together with a success. Apart from the window, the adapter is the only state carried across tasks (Fig. 1).

We make three contributions. First, we formulate fully test-time adaptation for GUI agents by three constraints that deployment imposes (Sec. 2) and place the existing approaches to improving GUI agents against it (Tab. 1). Second, we propose SOLO, a minimal weight-space method for the setting, in which a judge and a proposer–verifier pair read each episode and the admitted episodes and relabeled prefixes train a small adapter by top-K self-distillation (Sec. 3). Third, on recurring task streams we build in WebArena, VisualWebArena and MobileWorld, with two open agents, SOLO

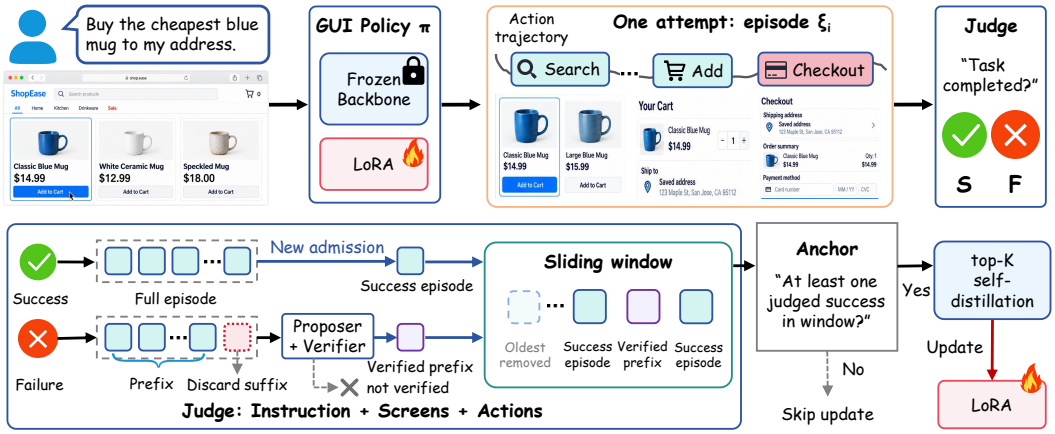


Figure 1: **Fully test-time adaptation for GUI agents, and SOLO.** Top: each task occurrence gets one attempt, and a judge decides from the instruction, screens and actions whether it was completed. Bottom: a judged success enters a sliding window as a full episode; for a judged failure, a proposer–verifier pair relabels a completed prefix, which enters the same window. While the window holds a judged success, one top-K self-distillation step updates the LoRA adapter.

improves on the frozen agent by three to six points of success rate and exceeds two in-setting memory methods on the web streams (Sec. 4).

2 SETTING: FULLY TEST-TIME ADAPTATION FOR GUI AGENTS

Deployment stream. A deployed agent faces tasks $\tau_1, \tau_2, \ldots$ in an order it does not control. A task $\tau = (g, e)$ pairs a natural-language instruction g with the environment state e in which it is issued. We call each position in the stream an occurrence. Tasks that share a template (the same errand with different arguments) recur along the stream, and some exact instances recur as well. For each occurrence the agent π_θ produces one episode $\xi = (o_0, y_0, o_1, y_1, \ldots, o_H)$ by observing the screen o_t, emitting a response y_t that contains its reasoning and an action, and continuing until it declares the task done or exhausts the step budget $B(\tau)$. Actions take effect in the environment, some of them irreversibly. There is no separate evaluation set: the stream is the evaluation, and every episode counts.

What is withheld, what is permitted. Fully test-time adaptation for GUI agents is the problem of improving π_θ along this stream under three constraints. *No ground truth*: no labels, rewards, demonstrations, or human feedback are available for any task after deployment, and the evaluator that scores the stream is invisible to the learner. *One attempt per occurrence*: the agent produces exactly one episode for each τ_i, in order, and that episode is the one that counts. It may not retry, sample alternatives, branch or reset the environment, and it may not practice on tasks before they count. *Deployment only*: the backbone is an off-the-shelf checkpoint, no training phase precedes or interleaves the stream, and whatever learning happens draws only on episodes the agent has already produced. Everything else is permitted. In particular, any signal that automated models can compute from the agent's own episodes is admissible, such as a model judging whether an episode completed its task, and any state may persist across the stream, whether an external memory or the agent's own weights.

Protocol. Because the stream is the evaluation, performance is measured prequentially: each episode is scored as it is produced, and the score is the benchmark's own success criterion, which the learner never sees. The quantity of interest is the difference in success rate between the adapting agent and the same agent frozen, both run on the identical stream in the identical order. Success over the stream and its profile across rounds are the reported quantities.

In-setting neighbors. Stream-based memory systems already satisfy the three constraints (Tab. 1, Sec. 5), and we compare against two of them. Workflow memory in its online form (Wang et al., 2024) processes test queries as a single stream and induces a workflow whenever a model-based evaluator judges the episode a success, and Darwinian Memory (Mi et al., 2026) runs the same tasks

for several rounds while a self-verifier prunes and reinforces a memory of sub-task trajectories. Both keep their experience in an external store and bring it back into the context or replay it. The agent's weights never change. SOLO occupies the other admissible choice of persistent state, the weights.

Relation to test-time adaptation of perception models. Tent defined fully test-time adaptation for perception models by withholding the labels and the training phase (Wang et al., 2021): the model arrives already trained, may not revisit the data it was trained on or change how it was trained, and is left alone with its test stream. Two of the three constraints above are its analogues: no ground truth, and no phase other than deployment. The third has none: running a perception model on an input has no consequence and can be repeated after an update, whereas an occurrence here is acted upon once and cannot be replayed. The one-attempt constraint encodes this difference. In both settings a learning signal can only be computed from what the model has already produced, but what it produces differs, and so does the signal. For a perception model it is the prediction it has just made, which is why minimizing its entropy, or training on its own confident predictions, are the canonical test-time signals. For a GUI agent it is the episode it has just produced, and the counterpart signal is a reading of that episode: an automated model can estimate from the episode alone whether it completed its task, without ground truth and without a second attempt, so this signal is admissible here for the same reason confidence is admissible there. Which signal to compute, and how to use it, is a method choice (Sec. 3).

3 METHOD: SOLO

SOLO adapts a small low-rank adapter along the stream (Fig. 1, Alg. 1). Tasks are handled strictly in arrival order. For each τ_i the agent produces one episode ξ_i under its current adapter, and that episode is what the stream records. Three auxiliary models then read ξ_i: a judge decides whether the task was completed, and for a judged failure a proposer and a verifier decide whether some prefix of the episode completed a subtask of its own. A judged success is admitted as it stands, and a verified prefix is admitted under the subtask's instruction. Admitted episodes enter a sliding window of the W most recent admissions, and each admission triggers one gradient step of top-K self-distillation on the window, applied to the adapter, provided the window holds at least one judged success. Apart from the window, the adapter is the only state carried from one task to the next.

Signals and relabeling. None of the three auxiliary models generates an action or a token target. We write a_t for the action contained in the response y_t and $\xi_{:k} = (o_0, y_0, \ldots, o_k, y_k)$ for the prefix of an episode through step k. The judge $J(g, \xi) \in \{0, 1\}$ reads the instruction with the episode's screens and actions and decides whether the task was completed (Pan et al., 2024). For a judged failure, the proposer $G(g, \xi)$ reads the full episode and either abstains or names an instruction g' that some prefix completed, together with the step k at which it was completed, as in hindsight relabeling (Andrychowicz et al., 2017; Zhang et al., 2023). The proposal is constrained: g' must be a subtask, prerequisite or narrower version of g about the same objects, and must be non-trivial and end on a resulting page or state. Guards apply before verification: the prefix must hold at least two actions and must not be dominated by repeated actions, since a prefix that contains the agent's loops would teach the loops. The verifier $V(g', a_{0:k}, o_{k+1})$ then checks the claim independently, seeing only g', the actions of the prefix and the screen that followed its last action, and accepts only if that screen shows g' completed. An accepted prefix $\xi_{:k}$ is admitted under g' as an ordinary episode: it enters the same window with the same weight and is trained with the same objective over the same positions, reasoning included. The branch is anchored: the window trains only while it holds at least one judged success, so a relabeled prefix is trained only in a window that also holds a judged success, and a prefix that leaves the window before a success arrives is never trained. The judge thus decides what the agent learns from, and the proposer and the verifier decide under which instruction a failed prefix may count.

Objective: top-K self-distillation. For an admitted episode ξ with supervised positions $\mathcal{M}(\xi)$, let $\bar{\theta}$ denote the pre-update parameters with gradients stopped, $\mathcal{V}_K(t)$ the K most probable tokens under $p_{\bar{\theta}}(\cdot \mid \xi_{<t})$, and

$$\mathcal{L}_K(\theta; \xi) \;=\; -\sum_{t \in \mathcal{M}(\xi)} \sum_{v \in \mathcal{V}_K(t)} q_t(v)\, \log p_\theta(v \mid \xi_{<t}), \qquad q_t(v) \;=\; \frac{p_{\bar{\theta}}(v \mid \xi_{<t})}{\sum_{u \in \mathcal{V}_K(t)} p_{\bar{\theta}}(u \mid \xi_{<t})}. \quad (1)$$

Algorithm 1 SOLO: fully test-time adaptation for a GUI agent

Require: frozen backbone θ_0; adapter $\phi \leftarrow 0$; judge J; proposer G; verifier V; window size W; support size K; step size η

$\mathcal{E} \leftarrow$ empty FIFO of capacity W
for each task $\tau_i = (g_i, e_i)$ in arrival order **do**
  $\xi_i \leftarrow \textsc{Rollout}(\pi_{\theta_0 \oplus \phi}, \tau_i)$ ▷ one attempt; its outcome is the evaluation
  $b_i \leftarrow J(g_i, \xi_i)$ ▷ automated success judgment
  if $b_i = 1$ **then**
    push ξ_i onto $\mathcal{E}$
  else
    $(g', k) \leftarrow G(g_i, \xi_i)$ ▷ subtask completed by a prefix, or $\perp$
    if $(g', k) \neq \perp$ **and** $V(g',\, a_{0:k},\, o_{k+1})$ **then**
      push $(g', \xi_{i,:k})$ onto $\mathcal{E}$ ▷ relabeled prefix
    end if
  end if
  if an episode was pushed **and** $\mathcal{E}$ holds a judged success **then**
    $\phi \leftarrow \phi - \eta \nabla_\phi \frac{1}{|\mathcal{E}|} \sum_{\xi \in \mathcal{E}} \mathcal{L}_K(\theta_0 \oplus \phi; \xi)$ ▷ Eq. 1, one gradient step; anchored
  end if
end for

The supervised positions are the tokens of the agent's own responses, reasoning and action alike, each conditioned on the observations and responses before it. The targets q_t come from the same forward pass with gradients stopped, so no second model and no cached teacher is involved. First, the signals select episodes, not tokens. A judged success is a verdict on the episode's outcome, and the objective never treats the executed token as correct: the target at every position is the truncated distribution of the pre-update policy itself. Second, the gradient with respect to the logits at a position is $p_\theta - q_t$ on the support and p_θ off it, which at the pre-update point is proportional to the tail mass $1 - \sum_{v \in \mathcal{V}_K(t)} p_{\bar{\theta}}(v)$. Each step therefore moves probability from the tail onto the policy's own top-K candidates, in proportion to their current probabilities and preserving their order. Every response position is in the loss, but a position where the policy is already near-deterministic has almost no tail mass and contributes almost nothing, so the update concentrates where the policy is uncertain. Third, K interpolates between two extremes. One-hot imitation of the executed tokens, the standard behavior-cloning loss, is not a self-distillation: its target puts all the probability on the executed token, so its gradient vanishes only when the policy has no alternative left at that position, and each update suppresses the alternatives the policy would need where the recorded action does not fit. In our ablation the one-hot variant's episodes lengthen round by round, and by the third round three quarters of its WebArena episodes run to the step budget (Sec. 4.3). At the other extreme, $K = |\mathcal{V}|$ is a no-op. A small K keeps the update a self-distillation that leaves the policy's own ranking intact.

Update. All adaptation lives in a small adapter ϕ on the frozen backbone θ_0, initialized with zero output (App. B) so that the deployed policy $\pi_{\theta_0 \oplus \phi}$ starts identical to the frozen agent and everything it becomes is attributable to the stream. Each admission triggers one gradient step on the mean of Eq. 1 over the window, at a small step size. Each episode takes part in at most W updates and then leaves the window, and nothing is sampled from older episodes. Per task, the cost is one judge call, at most two further calls for a judged failure, and at most one gradient step over W episodes, and the adapter is about 0.2% of the backbone.

4 EXPERIMENTS

4.1 SETUP AND PROTOCOL

Streams. We build three recurring streams, one per benchmark, and run every method on the identical stream in the identical order. The WebArena stream holds 108 task instances from 27 templates across four sites (an e-commerce store, its admin panel, a GitLab instance and a Reddit-style forum). The VisualWebArena stream holds 137 instances from 39 templates across a shopping site, a classifieds site and a forum. The MobileWorld stream holds 40 tasks spanning Gmail, Mastodon, device settings and native Android apps. Each stream visits its instances three times, once per round, for 324, 411 and 120 episodes. Each round is a seeded permutation of the instances, and the same stream file serves every run (App. C). The environment is restored to its initial state at every round boundary

Table 2: **Main result.** Success rate (%) over the stream under the benchmark's own evaluator; mean ± std over three runs per cell. Rows per stream: WebArena 324, VisualWebArena 411, MobileWorld 120. AWM-online and DMS are the in-setting memory baselines of Sec. 2, run on the identical streams.

	WebArena		VisualWebArena		MobileWorld	
Method	**UI-TARS-7B**	**Qwen3-VL-8B**	**UI-TARS-7B**	**Qwen3-VL-8B**	**UI-TARS-7B**	**Qwen3-VL-8B**
Frozen agent	20.8 ± 0.6	20.1 ± 0.8	16.7 ± 1.4	14.8 ± 0.6	6.1 ± 1.0	9.7 ± 1.0
AWM-online	22.7 ± 1.2	18.7 ± 1.6	16.3 ± 1.3	14.7 ± 1.2	$\mathbf{9.7 \pm 1.0}$	10.8 ± 1.7
DMS	22.4 ± 0.6	19.2 ± 0.5	17.0 ± 0.9	15.7 ± 0.9	7.5 ± 0.8	9.7 ± 0.5
SOLO (ours)	$\mathbf{25.8 \pm 1.6}$	$\mathbf{24.9 \pm 0.9}$	$\mathbf{20.0 \pm 0.8}$	$\mathbf{20.9 \pm 1.9}$	9.2 ± 0.8	$\mathbf{13.3 \pm 2.2}$

on the web and before every task on mobile, so a later round cannot pass by inheriting the state an earlier one left behind. The step budget is 30 on the web streams and 40 on MobileWorld.

Agents. We adapt two open GUI agents, UI-TARS-7B (Qin et al., 2025) and Qwen3-VL-8B (Bai et al., 2025), from their released checkpoints with their own action spaces and no training before the stream. Their prompts follow each release, with two notes shared by every method on the web streams, and are the benchmark's own on MobileWorld (App. G). Each emits a thought and one action per step.

SOLO configuration. The adapter is a rank-16 LoRA on the output and MLP projections of the upper half of the decoder layers, zero-initialized. The window holds $W{=}4$ episodes, the support is $K{=}4$, and each admission takes one AdamW step at learning rate 10^{-5}. The judge, the proposer and the verifier are all gpt-5-mini, used without fine-tuning; Tab. 4 swaps them for other models. A relabeled prefix must hold at least two actions with a repeat rate of at most 0.3. Prompts are given in App. G and the remaining details in App. B.

Baselines. The frozen agent is the same checkpoint run on the same stream without updates. AWM-online (Wang et al., 2024) and Darwinian Memory (DMS) (Mi et al., 2026) are the two in-setting memory methods of Sec. 2, adapted from the authors' released code to our agents and harness, with the same judge as their success signal. Both are training-free and keep what they learn in the prompt, so they differ from SOLO in what persists. The deviations from the released code are listed in App. B.

Metric. We report success rate over the stream under each benchmark's own evaluator. Every method and variant is run three times, and the tables give the mean and standard deviation. The runs behind every cell are listed in App. D.

4.2 MAIN RESULT: ADAPTATION IS FEASIBLE

Tab. 2 gives the main result. On every stream and with both agents, SOLO's success rate exceeds the frozen agent's by three to six points, and in every cell the margin exceeds the sum of the two standard deviations. The setting is therefore feasible as posed: an agent can improve its own weights during deployment from its own episodes, as read by auxiliary models, with no ground truth and one attempt per task occurrence.

The two memory baselines occupy the other admissible form of persistent state, and they read the same judge. On the web streams they track the frozen agent within about two points in either direction, whereas SOLO exceeds the better of the two by three to six points with both agents. On MobileWorld, where the frozen agents complete fewer than one task in ten, the picture is closer: AWM-online matches SOLO with UI-TARS-7B, at 9.7 against 9.2 with overlapping standard deviations, and trails it with Qwen3-VL-8B, at 10.8 against 13.3, while DMS stays near the frozen agent with both. On these streams, then, SOLO exceeds both memory methods on the web and is comparable to AWM-online on mobile.

4.3 ABLATIONS

Tab. 3 removes one component at a time from the full method on the Qwen3-VL-8B agent and both web streams: every variant stays above the frozen agent, and every removal lowers the mean over the two streams. Hindsight relabeling matters most: the success branch alone keeps about a third

Table 3: **Ablations.** One component removed at a time from the full method, on the Qwen3-VL-8B agent: the window, replaced by one step per admitted episode; top-K self-distillation, replaced by one-hot imitation of the executed tokens; and hindsight relabeling, leaving the success branch alone. Success rate (%), mean $\pm$ std over three runs, and the mean over the two streams.

Variant	VisualWebArena	WebArena	Mean
Frozen agent	14.8 ± 0.6	20.1 ± 0.8	17.5
SOLO (full)	20.9 ± 1.9	24.9 ± 0.9	22.9
w/o window (one step per episode)	18.2 ± 1.2	23.0 ± 1.2	20.6
w/o top-K self-distillation (one-hot targets)	20.6 ± 2.6	22.0 ± 1.5	21.3
w/o hindsight relabeling (success branch only)	17.0 ± 1.4	21.9 ± 0.6	19.4

Table 4: **Dependence on the auxiliary models.** SOLO on the Qwen3-VL-8B agent with the judge, the proposer and the verifier all replaced by the same alternative model. Success rate (%), mean $\pm$ std over three runs, and the judge's precision, the share of the episodes it admitted that the benchmark evaluator scores as successes, averaged over the runs.

	VisualWebArena		WebArena	
Auxiliaries	**Success (%)**	**Judge prec.**	**Success (%)**	**Judge prec.**
Frozen agent (no auxiliaries)	14.8 ± 0.6	–	20.1 ± 0.8	–
gpt-5-mini	20.9 ± 1.9	0.51	24.9 ± 0.9	0.66
gpt-5	18.9 ± 1.4	0.59	23.0 ± 0.7	0.76
gpt-5-nano	19.1 ± 0.6	0.43	23.9 ± 1.7	0.46
the agent's own model (Qwen3-VL-8B)	17.3 ± 0.8	0.35	24.2 ± 1.0	0.54

of the margin, and it updates on about a quarter of the episodes on VisualWebArena and a fifth on WebArena, whereas the verified prefixes raise both shares to about half (Tab. 8). The window comes next: one step per admitted episode keeps a little over half of the margin, and its episodes stay as short as the full method's. Replacing top-K self-distillation with one-hot imitation of the executed tokens leaves VisualWebArena level and costs three points on WebArena. The one-hot gradient is more than an order of magnitude larger at the same learning rate (App. D) and pushes every alternative toward zero, and our reading is that the policy over-commits to its recorded action sequences. Its episodes lengthen round by round (Tab. 10): by the third round about three quarters of them on WebArena run to the step budget, against one in five for the full method, and its success rate there falls back to the frozen level. On VisualWebArena the episodes lengthen in the same way but the wins hold within three rounds, so on these streams the target form matters only on WebArena, whose episodes are longer.

4.4 SENSITIVITY TO THE AUXILIARY MODELS

Tab. 4 replaces the judge, the proposer and the verifier together with three alternatives to gpt-5-mini. With gpt-5 or gpt-5-nano the success rate stays within two points on both streams, and every choice, including the agent's own 8B model, keeps the method above the frozen agent. SOLO is therefore robust to the choice of auxiliary models. The one row that separates is the agent's own model on VisualWebArena, at 3.6 points below gpt-5-mini. It is also the row with the lowest judge precision, about a third of the admitted episodes being true successes, whereas precision otherwise does not order the rows: gpt-5 is the most precise judge on both streams and does not give the highest success rate. Our reading is that the judge acts as a soft filter rather than as a stand-in for the evaluator. It admits the episodes that look like completions, and those carry a usable signal even where the evaluator would reject some of them, until precision falls far enough that the admitted set no longer resembles success.

4.5 WHERE THE GAIN APPEARS

Fig. 2 plots the cumulative success rate along each stream, and Tab. 7 gives the per-round counts behind it. Read by round, the margin over the frozen agent takes two shapes. With Qwen3-VL-8B on WebArena and UI-TARS-7B on VisualWebArena it is present from the first round, before any

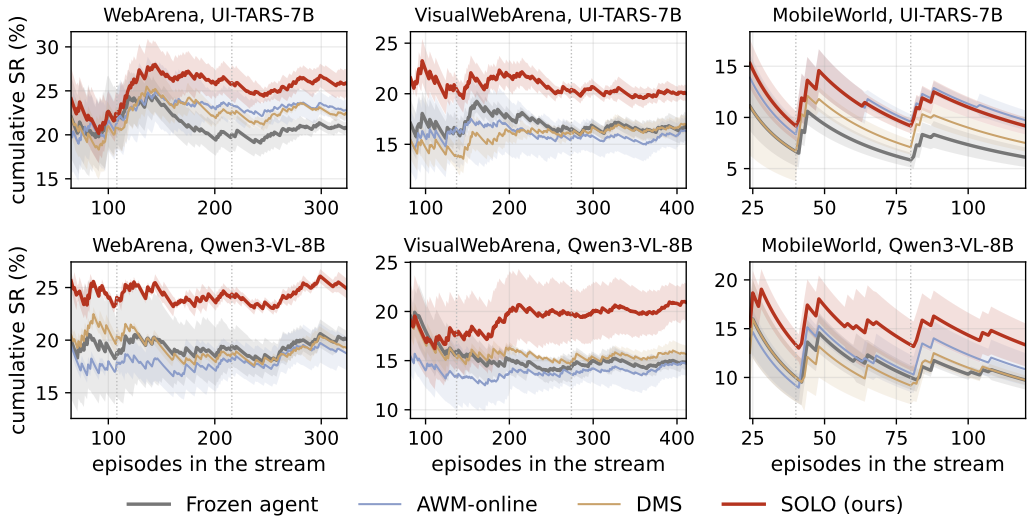


Figure 2: **Cumulative success rate along the stream.** Each panel is one cell of Tab. 2: the success rate over the episodes seen so far, averaged over the three runs, with a band of one standard deviation across the runs. Dotted lines mark the round boundaries. Each panel starts at twenty percent of its stream, where the earlier points rest on too few episodes.

instance recurs, although templates already recur within that round (App. C). With UI-TARS-7B on WebArena and Qwen3-VL-8B on VisualWebArena the first round ends within three wins of the frozen agent, and the margin opens in the second and third rounds, when the instances recur. On MobileWorld the per-round margins are one to two tasks of forty in every round and do not separate the two shapes. We read this as adaptation to the deployment distribution rather than memorization of particular tasks. As in fully test-time adaptation for perception models, where the model adjusts to the distribution of its test inputs, the agent here adjusts to the sites and errands of its stream, and the benefit shows in its online performance.

5 Related Work

Test-time adaptation. Test-time adaptation updates a trained model on unlabeled test inputs as they arrive (Liang et al., 2024). Tent minimizes prediction entropy and touches only normalization parameters (Wang et al., 2021), test-time training adds a self-supervised head but must alter training to do so (Sun et al., 2020). The continual variants guard against drift over a long stream, CoTTA by distilling toward a weight-averaged teacher and restoring weights to the source (Wang et al., 2022), EATA by selecting reliable samples and regularizing toward the source (Niu et al., 2022). SOLO relates to these at three points: its judge plays the role that confidence plays in EATA's sample selection, its top-K self-distillation is the analogue of CoTTA's soft pseudo-labels, taken from the same forward pass rather than from a separate teacher, and its update of a small adapter parallels Tent's restriction to normalization layers. The difference lies in the signal: a confident prediction is at once evidence and target, whereas an episode-level verdict names no action.

Adaptation through extra rollouts or a training phase. Language models are adapted at test time by fine-tuning on data derived from the test instance (Akyürek et al., 2025) or by treating the majority vote over many samples as a reward (Zuo et al., 2025). For agents the extra compute goes into rollouts: GTA1 samples several candidate actions and lets a judge pick one (Yang et al., 2026), and JIT-RL, EvoTest and GTTA spend several attempts or exploration episodes on each task or environment (Li et al., 2026a; He et al., 2026; Chen et al., 2026a). Online reinforcement learning spends a training phase instead, with an environment evaluator or a trained reward model, resets and parallel rollouts (Bai et al., 2024; Qi et al., 2025; Wang et al., 2025a; Chen et al., 2026b; Gu et al., 2026), and the flywheels behind self-evolving agents iterate that loop with reward models, synthesized tasks or rejection fine-tuning (Gao et al., 2026a; Xiao et al., 2025; Lin et al., 2026; Li et al., 2026b; Zhai et al., 2025; Jin et al., 2026). Each relies on a resource the setting withholds: samples or attempts

beyond the one that counts, or a training phase with rewards. SOLO also adapts from the agent's own experience, but from one attempt per occurrence and without a reward.

Experiential memory. Agents also improve by keeping experience in an external store. Reflexion and ExpeL reflect on failed attempts and try again, or distill insights from a training split (Shinn et al., 2023; Zhao et al., 2024), MAGNET harvests trajectories over passes through the tasks before they count (Sun et al., 2026), and EchoTrail fills its memory in an exploration stage (Li et al., 2025): these are practice-based and use rollouts that the one-attempt constraint withholds. Stream-based systems learn as they go, admitting entries under a model judge or reflector, AWM-online by inducing workflows (Wang et al., 2024; Pan et al., 2024), Darwinian Memory by scoring and pruning (Mi et al., 2026), ReasoningBank by distilling strategies (Ouyang et al., 2026), and Mobile-Agent-E by evolving tips (Wang et al., 2025b). These operate in our setting with memory as the persistent state, the other admissible choice. SOLO takes the weights instead, and Sec. 4.2 compares it with two of these systems on the same streams with the same judge.

Self-imitation and learning from failures. The objective relates to self-imitation and rejection sampling fine-tuning, which train on the agent's own outputs that pass a check (Oh et al., 2018; Zelikman et al., 2022; Singh et al., 2024; Yuan et al., 2023), with LLM judges standing in for the check on GUI trajectories (Pan et al., 2024; Xiao et al., 2025). SOLO applies that selection online, one episode at a time, and distills toward the policy's own truncated distribution. The failure branch relates to work on learning from failed trajectories: hindsight relabeling turns a failure into a success for the goal it reached (Andrychowicz et al., 2017; Zhang et al., 2023; Ding, 2026), inverse-dynamics objectives and early experience learn from the transitions an agent's own actions produce, without a reward (Gao et al., 2026b; Zhang et al., 2026), and unlikelihood or preference losses train against the failed actions themselves (Welleck et al., 2020; Ethayarajh et al., 2024). SOLO takes the first route: it relabels a verified prefix with the subtask it completed and admits it through the same update.

6 DISCUSSION AND LIMITATIONS

Within three rounds of a recurring stream, an off-the-shelf agent improves its own weights from its deployment episodes, read by auxiliary models, with no ground truth and one attempt per task occurrence, on three benchmarks and with two agents. The claim concerns online performance on that stream: as with Tent, which adapts a model to the test distribution itself, the agent adapts to the sites and errands it is deployed on. The streams are built from benchmark tasks that recur three times, not from user logs, so how often a real deployment repeats an errand, and hence the size of the gain there, is not measured. On MobileWorld the frozen agents pass fewer than one task in ten, so judged successes are rare.

The signal depends on auxiliary models: each episode costs one judge call and each judged failure at most two more. With the agent's own 8B model in every auxiliary role the gain holds on WebArena and shrinks on VisualWebArena (Tab. 4), and swapping either role alone accounts for most of that loss (App. E). Our reading is that the judge acts as a soft filter: it reads the episode and not the environment, it admits what looks like a completion on screen, and such episodes carry a usable signal even when the evaluator would reject some of them, which is consistent with precision not ordering the rows of Tab. 4 and with the signal fading only when the admitted set stops resembling success. The same channel is an attack surface: page content that sways the judge or the proposer trains the adapter, and we do not study adversarial pages.

Under the same judge, SOLO exceeded two prompt-memory methods on the web streams and was comparable to AWM-online on mobile (Tab. 2). This is evidence for the in-weight route rather than a verdict between the two: memory keeps experience in an inspectable, deletable store and adapts in context, weights internalize it, and recent position papers argue that the two are complementary and that durable adaptation needs its own channel beside retrieval (Xu et al., 2026; Dorovatas et al., 2026). The cost of persistence is drift. By the third round the one-hot variant runs three quarters of its WebArena episodes to the step budget, and the full method's episodes also lengthen with Qwen3-VL-8B, by two to three steps with the share ending on the budget rising to about one in six, while with UI-TARS-7B they shorten slightly (Tab. 10). Behavior over longer deployments is untested, and the setting provides no ground truth with which to detect degradation. Resetting the adapter to zero restores the frozen agent exactly, and detecting when to reset is not evaluated here.

AI USE STATEMENT

Large language and vision-language models appear in this work in two roles. First, as components of the experiments: the judge, the proposer and the verifier of SOLO are gpt-5-mini by default and gpt-5, gpt-5-nano or Qwen3-VL-8B in Sec. 4.4, the memory baselines use the same judge and, for AWM-online, the authors' gpt-4o induction model, and the VisualWebArena image-query evaluator is served by gpt-5-mini in place of BLIP-2. Every such use is described in Sec. 4.1 and App. B, App. C and App. G, with the prompts reproduced verbatim. Second, as tools: AI coding assistants helped write and debug the experiment harness and the analysis scripts, and an AI writing assistant helped edit the text of this paper under the authors' direction. All experiments were designed, run and checked by the authors, every number in the paper is a logged measurement from the runs listed in App. A, App. D and App. E, and the authors take full responsibility for the content.

REPRODUCIBILITY STATEMENT

The setting is defined in Sec. 2 and the method in Sec. 3 and Alg. 1. The configuration used in every run (adapter geometry, optimizer, window and support sizes, and the guards of the failure branch) is given in Sec. 4.1 and App. B, together with the deviations of the two adapted baselines from their released code. The three streams are fixed files: their composition, order, resets and evaluators are described in App. C, and every run behind every table is listed in App. D with its per-round wins. The prompts of the agents, the judge, the proposer, the verifier and the baselines are reproduced verbatim in App. G. The benchmarks are the public WebArena, VisualWebArena and MobileWorld releases, self-hosted. We will release the code, the stream files and the run logs behind every number.

ETHICS STATEMENT

All experiments ran on self-hosted copies of the benchmark websites and on Android emulators. No real user account, no real person's data and no live service was involved, and the screenshots sent to the auxiliary models show only these sandboxed environments. The setting studied here, an agent that updates its own weights while deployed, carries risks beyond those of a frozen agent. A GUI agent's actions can be irreversible, an agent that learns from what it sees is exposed to page content crafted to sway its judge or its proposer, and a self-updating policy can drift without any ground truth to detect it. Sec. 6 states these risks and the one safeguard the design leaves available, resetting the adapter, which restores the frozen agent exactly. We do not study adversarial pages, and we regard such a study as a prerequisite for any deployment of the method outside a sandbox.

REFERENCES

Ekin Akyürek, Mehul Damani, Adam Zweiger, Linlu Qiu, Han Guo, Jyothish Pari, Yoon Kim, and Jacob Andreas. The surprising effectiveness of test-time training for few-shot learning. In *International Conference on Machine Learning (ICML)*, 2025.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

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

Arthur Chen, Zuxin Liu, Jianguo Zhang, Akshara Prabhakar, Zhiwei Liu, Shelby Heinecke, Silvio Savarese, Victor Zhong, and Caiming Xiong. Test-time adaptation for LLM agents via environment interaction. In *International Conference on Learning Representations (ICLR)*, 2026a.

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Sara Cao, Yuanhao Xiong, Haibo Tong, Huaxiu Yao, Hengduo Li, Jiacheng Zhu, Xian Li, Dawn Song, Bo Li, Jason Weston, and Dat Huynh. Scaling agent learning via experience synthesis. In *International Conference on Learning Representations (ICLR)*, 2026b.

Liang Ding. AgentHER: Hindsight experience replay for LLM agent trajectory relabeling. *arXiv preprint arXiv:2603.21357*, 2026.

Vaggelis Dorovatas, Malte Schwerin, Andrew D. Bagdanov, Lucas Caccia, Antonio Carta, Laurent Charlin, Barbara Hammer, Tyler L. Hayes, Timm Hess, Christopher Kanan, Dhireesha Kudithipudi, Xialei Liu, Vincenzo Lomonaco, Jorge Mendez-Mendez, Darshan Patil, Ameya Prabhu, Elisa Ricci, Tinne Tuytelaars, Gido M. van de Ven, Liyuan Wang, Joost van de Weijer, Jonghyun Choi, Martin Mundt, and Rahaf Aljundi. Position: Modular memory is the key to continual learning agents. In *International Conference on Machine Learning (ICML)*, 2026.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. KTO: Model alignment as prospect theoretic optimization. *arXiv preprint arXiv:2402.01306*, 2024.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Qihan Ren, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026a.

Longxi Gao, Li Zhang, Pengzhi Gao, Wei Liu, Jian Luan, and Mengwei Xu. GUI-Shift: Enhancing VLM-based GUI agents through self-supervised reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026b.

Li Gu, Zihuan Jiang, Zhixiang Chi, Huan Liu, Ziqiang Wang, Yuanhao Yu, Glen Berseth, and Yang Wang. Generalization in online reinforcement learning for mobile agents. *arXiv preprint arXiv:2603.07432*, 2026.

Yufei He, Juncheng Liu, Yue Liu, Yibo Li, Tri Cao, Zhiyuan Hu, Xinxing Xu, and Bryan Hooi. EvoTest: Evolutionary test-time learning for self-improving agentic systems. In *International Conference on Learning Representations (ICLR)*, 2026.

Shilong Jin, Lanjun Wang, and Zhuosheng Zhang. SE-GA: Memory-augmented self-evolution for GUI agents. In *International Conference on Machine Learning (ICML)*, 2026.

Runze Li, Yuwen Zhai, Bo Xu, Liwu Xu, Nian Shi, Wei Zhang, Ran Lin, and Liang Wang. EchoTrail-GUI: Building actionable memory for GUI agents via critic-guided self-exploration. *arXiv preprint arXiv:2512.19396*, 2025.

Yibo Li, Zijie Lin, Ailin Deng, Xuan Zhang, Yufei He, Shuo Ji, Tri Cao, and Bryan Hooi. Just-in-time reinforcement learning: Continual learning in LLM agents without gradient updates. *arXiv preprint arXiv:2601.18510*, 2026a.

Zecheng Li, Zhihui Cao, Wenke Huang, Yudong Zhang, Keying Qi, Rui Wang, Zeyu Zheng, Jian Zhao, Hao Zhu, Hengxin Wu, Yuran Wang, Guitao Fan, Guokun Wu, Yicong Liu, Zhilin Gao, Haikun Xu, He Yang, Minqi Xiang, Xingyu Liu, and Zuojian Wang. MagicGUI-RMS: A multi-agent reward model system for self-evolving GUI agents via automated feedback reflux. *arXiv preprint arXiv:2601.13060*, 2026b.

Jian Liang, Ran He, and Tieniu Tan. A comprehensive survey on test-time adaptation under distribution shifts. *International Journal of Computer Vision*, 2024.

Zichuan Lin, Feiyu Liu, Yijun Yang, Jiafei Lyu, Yiming Gao, Yicheng Liu, Zhicong Lu, Yangbin Yu, Mingyu Yang, Junyou Li, Deheng Ye, and Jie Jiang. UI-Voyager: A self-evolving GUI agent learning via failed experience. *arXiv preprint arXiv:2603.24533*, 2026.

Zhaoyang Liu, Jingjing Xie, Zichen Ding, Zehao Li, Bowen Yang, Zhenyu Wu, Xuehui Wang, Qiushi Sun, Shi Liu, Weiyun Wang, Shenglong Ye, Qingyun Li, Xuan Dong, Yue Yu, Chenyu Lu, YunXiang Mo, Yao Yan, Zeyue Tian, Xiao Zhang, Yuan Huang, Yiqian Liu, Weijie Su, Gen Luo, Xiangyu Yue, Biqing Qi, Kai Chen, Bowen Zhou, Yu Qiao, Qifeng Chen, and Wenhai Wang. ScaleCUA: Scaling open-source computer use agents with cross-platform data. *arXiv preprint arXiv:2509.15221*, 2025.

Hongze Mi, Yibo Feng, Wenjie Lu, Song Cao, Jinyuan Li, Yanming Li, Xuelin Zhang, Haotian Luo, Songyang Peng, He Cui, Tengfei Tian, Jun Fang, Hua Chai, and Naiqiang Tan. Darwinian memory: A training-free self-regulating memory system for GUI agent evolution. *arXiv preprint arXiv:2601.22528*, 2026.

Shuaicheng Niu, Jiaxiang Wu, Yifan Zhang, Yaofo Chen, Shijian Zheng, Peilin Zhao, and Mingkui Tan. Efficient test-time model adaptation without forgetting. In *International Conference on Machine Learning (ICML)*, 2022.

Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. In *International Conference on Machine Learning (ICML)*, 2018.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations (ICLR)*, 2026.

Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. Autonomous evaluation and refinement of digital agents. *arXiv preprint arXiv:2404.06474*, 2024.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, Jie Tang, and Yuxiao Dong. WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2025.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

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

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Libo Sun, Jiwen Zhang, Siyuan Wang, and Zhongyu Wei. MAGNET: Towards adaptive GUI agents with memory-driven knowledge evolution. *arXiv preprint arXiv:2601.19199*, 2026.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei A. Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *International Conference on Machine Learning (ICML)*, 2020.

Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. Tent: Fully test-time adaptation by entropy minimization. In *International Conference on Learning Representations (ICLR)*, 2021.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, Wanjun Zhong, Yining Ye, Yujia Qin, Yuwen Xiong, Yuxin Song, Zhiyong Wu, Aoyan Li, Bo Li, Chen Dun, Chong Liu, Daoguang Zan, Fuxing Leng, Hanbin Wang, Hao Yu, Haobin Chen, Hongyi Guo, Jing Su, Jingjia Huang, Kai Shen, Kaiyu Shi, Lin Yan, Peiyao Zhao, Pengfei Liu, Qinghao Ye, Renjie Zheng, Shulin Xin, Wayne Xin Zhao, Wen Heng, Wenhao Huang, Wenqian Wang, Xiaobo Qin, Yi Lin, Youbin Wu, Zehui Chen, Zihao Wang, Baoquan Zhong, Xinchun Zhang, Xujing Li, Yuanfan Li, Zhongkai Zhao, Chengquan Jiang, Faming Wu, Haotian Zhou, Jinlin Pang, Li Han, Qi Liu, Qianli Ma, Siyao Liu, Songhua Cai, Wenqi Fu, Xin Liu, Yaohui Wang, Zhi Zhang, Bo Zhou, Guoliang Li, Jiajun Shi, Jiale Yang, Jie Tang, Li Li, Qihua Han, Taoran Lu, Woyu Lin, Xiaokang Tong, Xinyao Li, Yichi Zhang, Yu Miao, Zhengxuan Jiang, Zili Li, Ziyuan Zhao, Chenxin Li, Dehua Ma, Feng Lin, Ge Zhang, Haihua Yang, Hangyu Guo, Hongda Zhu, Jiaheng Liu, Junda Du, Kai Cai, Kuanye Li, Lichen Yuan, Meilan Han, Minchao Wang, Shuyue Guo, Tianhao Cheng, Xiaobo Ma, Xiaojun Xiao, Xiaolong Huang, Xinjie Chen, Yidi Du, Yilin Chen, Yiwen Wang, Zhaojian Li, Zhenzhu Yang, Zhiyuan Zeng, Chaolin Jin, Chen Li, Hao Chen, Haoli Chen, Jian Chen, Qinghao Zhao, and Guang Shi. UI-TARS-2 technical report: Advancing GUI agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025a.

Qin Wang, Olga Fink, Luc Van Gool, and Dengxin Dai. Continual test-time domain adaptation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

Zhenhailong Wang, Haiyang Xu, Junyang Wang, Xi Zhang, Ming Yan, Ji Zhang, Fei Huang, and Heng Ji. Mobile-Agent-E: Self-evolving mobile assistant for complex tasks. *arXiv preprint arXiv:2501.11733*, 2025b.

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

Sean Welleck, Ilia Kulikov, Stephen Roller, Emily Dinan, Kyunghyun Cho, and Jason Weston. Neural text generation with unlikelihood training. In *International Conference on Learning Representations (ICLR)*, 2020.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and Yu Qiao. OS-ATLAS: Foundation action model for generalist GUI agents. In *International Conference on Learning Representations (ICLR)*, 2025.

Han Xiao, Guozhi Wang, Yuxiang Chai, Zimu Lu, Weifeng Lin, Hao He, Lue Fan, Liuyang Bian, Rui Hu, Liang Liu, Shuai Ren, Yafei Wen, Xiaoxin Chen, Aojun Zhou, and Hongsheng Li. UI-Genie: A self-improving approach for iteratively boosting MLLM-based mobile GUI agents. *arXiv preprint arXiv:2505.21496*, 2025.

Binyan Xu, Xilin Dai, and Kehuan Zhang. Contextual agentic memory is a memo, not true memory. *arXiv preprint arXiv:2604.27707*, 2026.

Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, Ran Xu, Liyuan Pan, Caiming Xiong, and Junnan Li. GTA1: GUI test-time scaling agent. In *International Conference on Learning Representations (ICLR)*, 2026.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*, 2023.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Yunpeng Zhai, Shuchang Tao, Cheng Chen, Anni Zou, Ziqian Chen, Qingxu Fu, Shinji Mai, Li Yu, Jiaji Deng, Zouying Cao, Zhaoyang Liu, Bolin Ding, and Jingren Zhou. AgentEvolver: Towards efficient self-evolving agent system. *arXiv preprint arXiv:2511.10395*, 2025.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. AppAgent: Multimodal agents as smartphone users. In *CHI Conference on Human Factors in Computing Systems (CHI)*, 2025.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, Jian Xie, Yuxuan Sun, Boyu Gou, Qi Qi, Zihang Meng, Jianwei Yang, Ning Zhang, Xian Li, Ashish Shah, Dat Huynh, Hengduo Li, Zi Yang, Sara Cao, Lawrence Jang, Shuyan Zhou, Jiacheng Zhu, Huan Sun, Jason Weston, Yu Su, and Yifan Wu. Agent learning via early experience. In *International Conference on Machine Learning (ICML)*, 2026.

Tianjun Zhang, Fangchen Liu, Justin Wong, Pieter Abbeel, and Joseph E. Gonzalez. The wisdom of hindsight makes language models better instruction followers. In *International Conference on Machine Learning (ICML)*, 2023.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *AAAI Conference on Artificial Intelligence*, 2024.

Yuxin Zuo, Kaiyan Zhang, Li Sheng, Shang Qu, Ganqu Cui, Xuekai Zhu, Haozhan Li, Yuchen Zhang, Xinwei Long, Ermo Hua, Biqing Qi, Youbang Sun, Zhiyuan Ma, Lifan Yuan, Ning Ding, and Bowen Zhou. TTRL: Test-time reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

A SIGNAL STUDY: ENTROPY MINIMIZATION

The canonical unsupervised signal of test-time adaptation for perception models is the model's own confidence: Tent adapts by minimizing the entropy of its predictions (Wang et al., 2021). Tab. 5 applies the same signal to a GUI agent. This variant keeps SOLO's adapter, window and step size and replaces the loss with the entropy of the agent's next-token distribution, averaged over the positions of its own response, applied to every episode, with no judge and no relabeling. The agent is Qwen3-VL-8B on both web streams, three runs each.

Entropy minimization does not help on either stream, and on VisualWebArena it collapses the agent. The response entropy falls by an order of magnitude along the stream in every run. On VisualWebArena the policy stops acting: episodes that end at their first step, with an answer or a declaration of completion, rise from about a quarter of a round under the frozen agent to between half and most of the third round, and the wins fall from the frozen level in the first round to a handful in the third. On WebArena the entropy falls in the same way, but the agent keeps acting and its success rate stays at the frozen level. Sharpening the policy toward what it already prefers therefore either collapses it or leaves it where it was, and the signal of SOLO comes from a judge instead.

Table 5: **Entropy minimization as the test-time signal.** Qwen3-VL-8B, both web streams, three runs each, against the frozen agent. Wins per round with the total, episodes ended at their first step per round, and the mean per-episode response entropy in nats over the first quarter of the stream and over the last.

Stream	Arm	Wins per round (total)	One-step episodes	Entropy
VisualWebArena	frozen (mean of 3 runs)	22 / 18 / 21 (61)	36 / 35 / 33	–
	entropy min., run 1	18 / 4 / 1 (23)	55 / 90 / 117	0.116 → 0.014
	entropy min., run 2	22 / 10 / 4 (36)	48 / 53 / 83	0.113 → 0.012
	entropy min., run 3	20 / 6 / 8 (34)	48 / 87 / 76	0.114 → 0.014
WebArena	frozen (mean of 3 runs)	20 / 22 / 23 (65)	5 / 6 / 6	–
	entropy min., run 1	21 / 24 / 21 (66)	6 / 2 / 2	0.111 → 0.016
	entropy min., run 2	22 / 19 / 17 (58)	3 / 3 / 5	0.107 → 0.015
	entropy min., run 3	24 / 24 / 22 (70)	6 / 8 / 18	0.113 → 0.014

B METHOD DETAILS

Objective and gradient. For a position t with logits z_t, the gradient of Eq. 1 is $\partial \mathcal{L}_K / \partial z_t = p_\theta(\cdot \mid \xi_{<t}) - \tilde{q}_t$, where $\tilde{q}_t$ equals q_t on the support $\mathcal{V}_K(t)$ and zero off it. At the pre-update point, with $S_t = \sum_{v \in \mathcal{V}_K(t)} p_{\bar{\theta}}(v \mid \xi_{<t})$ the mass of the support, the component on a support token v is $-p_{\bar{\theta}}(v)(1 - S_t)/S_t$ and the component on any other token is $p_{\bar{\theta}}(v)$, so the gradient moves exactly the tail mass $1 - S_t$ from outside the support onto it, in proportion to the current probabilities, and vanishes where the support already holds all the mass. The targets come from the same forward pass with gradients stopped. The supervised positions are all response tokens of every step of the episode, reasoning and action alike. Each step is conditioned on the instruction, the screenshots seen so far and the earlier responses, exactly as it was served, and a VisualWebArena task's input images precede the screenshots. The loss is summed over an episode's supervised positions and divided by their number, then averaged over the episodes in the window, and one optimizer step follows. A relabeled prefix is trained with the same scope under its own instruction.

Adapter and optimizer. The adapter is a LoRA of rank 16 and scale $\alpha = 32$ without dropout on the attention output projection and the three MLP projections of the upper half of the decoder layers of the language model, with the vision encoder untouched. Its A matrices are drawn from $\mathcal{N}(0, 0.02)$ with a fixed seed and its B matrices are zero, so the adapted policy starts identical to the frozen agent. The optimizer is AdamW at learning rate 10^{-5} without weight decay, the gradient norm is clipped at one, and the model runs in bfloat16. The window holds $W = 4$ episodes and the support is $K = 4$. Every variant of Tab. 3 and Tab. 4 keeps these settings and changes only the named component, and both agents use the same values on every stream.

Failure branch. The proposer sees the original instruction and, for every step, the screenshot the agent saw, its reasoning and its action. It answers whether some prefix completed a subtask,

prerequisite or narrower version of the task, the earliest step k at which that was completed, and the instruction g' that names it, or abstains. Before verification the prefix up to k, with a trailing declaration of completion dropped, must hold at least two actions, and the share of its steps that repeat the previous action must not exceed 0.3. The verifier sees only g', the actions of the prefix and the screenshot after step k, and answers whether that screenshot shows g' completed. An episode that ended at its first step has no prefix and is skipped. An admitted prefix enters the window under g' with the same weight and the same loss scope as a judged success, and the anchor holds any update until the window holds a judged success. Tab. 6 gives the fate of the judged failures in the three runs of each cell. The guards are the largest filter, and most of what they reject is a proposed prefix of a single action. The verifier rejects between one in thirteen and one in four of the prefixes that reach it. The admitted prefixes hold three to four actions on average. On WebArena about half of the failures end as admitted prefixes, on MobileWorld about three fifths, and on VisualWebArena about a quarter. The anchor withheld an update at about one admission in seven for Qwen3-VL-8B on WebArena, at about one in twenty for UI-TARS-7B on both web streams, and never on MobileWorld or for Qwen3-VL-8B on VisualWebArena.

Table 6: **What becomes of a judged failure.** Per stream and agent, mean over the three SOLO runs: episodes, judged successes, judged failures, and of the failures those with no prefix (ended at the first step), those on which the proposer abstained, those rejected by the guards (in parentheses, rejected as a single-action prefix), those rejected by the verifier, and those admitted as relabeled prefixes, with the admitted share of the failures and the mean number of steps in an admitted prefix.

Stream	Agent	Episodes	Succ.	Fail.	No prefix	Abstain	Guards	Verifier	Admitted	Prefix steps
WebArena	UI-TARS-7B	324	96	228	1	42	63 (53)	22	101 (44%)	3.7
WebArena	Qwen3-VL-8B	324	74	250	6	23	69 (51)	19	134 (54%)	3.0
VisualWebArena	UI-TARS-7B	411	94	317	2	67	133 (100)	24	91 (29%)	3.3
VisualWebArena	Qwen3-VL-8B	411	127	284	49	45	100 (83)	20	69 (24%)	3.0
MobileWorld	UI-TARS-7B	120	23	97	0	12	19 (13)	10	56 (58%)	3.9
MobileWorld	Qwen3-VL-8B	120	25	95	0	6	22 (16)	5	62 (65%)	3.6

Baselines. AWM-online follows the online loop of the authors' released code (Wang et al., 2024): after every judged success the workflow file of the site is re-induced from scratch over all judged-positive episodes so far, deduplicated by template with one example per template, with the authors' induction prompt verbatim, at temperature one, and the completion is the workflow file that the agent then sees in its prompt. Three deviations adapt the method to our agents and harness. The one-shot example and the appended action-space note are rewritten in our agents' action vocabulary, since the original agent acts on accessibility trees. The induction model stays the authors' gpt-4o, because gpt-5-mini returned empty inductions at the authors' token budget, while the judge that gates memory writes is gpt-5-mini as for SOLO. The rendered workflow block is capped at four thousand characters, since a 7B context cannot hold an unbounded library. Darwinian Memory follows the authors' released code (Mi et al., 2026), an anonymized repository linked from the paper that we accessed on 18 September 2026 and that has since expired: their memory score with their constants and logical clock, their strike semantics with deletion at three strikes, and their retrieval with rerank and top three, verified by the same judge. Darwinian Memory needs two deviations. The similarity is an IDF-weighted token cosine instead of a sentence-embedding model, with the admission threshold recalibrated from 0.8 to 0.15, which admits no cross-template pair and keeps 64% of same-template pairs on the WebArena intents. Their planner reputation manager regulates a planner's plans and has no counterpart in a single-screenshot policy. Both baselines keep the policy frozen and everything they learn in the prompt.

C STREAMS AND PROTOCOL

WebArena. The stream holds 108 instances of 27 WebArena templates, four per template: ten templates on the shopping site, nine on GitLab, four on the shopping admin panel and four on the forum.

VisualWebArena. The stream holds 137 instances of 39 VisualWebArena templates, three or four per template: 28 templates on the shopping site, nine on classifieds and two on the forum.

MobileWorld. MobileWorld tasks have no parameterized instances, so each task is one instance. The stream holds 40 tasks: sixteen on Mastodon, nine on Gmail, eight on native Android apps and seven on device settings.

Order and resets. Every stream has three rounds. Within a round the instances are permuted with a seed, and the permutation is redrawn until no instance recurs within eight rows of its previous visit, so the same order serves every run. On the web streams the sites are restored to their initial state at every round boundary through the benchmarks' reset procedures, since many templates change the state of a site and a later round would otherwise inherit what an earlier one left behind. On MobileWorld the benchmark's own setup and teardown restore the device before every task. The step budget is 30 on the web streams and 40, the official budget, on MobileWorld.

Evaluators. The evaluators are the benchmarks' own and are never visible to the agent or to the auxiliary models. VisualWebArena's image-query evaluator takes a visual question answering function, which gpt-5-mini serves in place of the reference implementation's BLIP-2. MobileWorld's tasks are graded by the benchmark's checkers.

D FULL RESULTS

Tab. 7 lists the three runs behind every cell of Tab. 2: total wins with the per-round split, the mean and std of wins, and the success rate. Tab. 8 lists the runs behind every row of Tab. 3 in the same form, and Tab. 9 lists every run behind Tab. 4 with the judge's precision on each.

Table 7: **Every run behind Tab. 2.** Wins per run with the round split in parentheses, then mean ± std of wins and success rate (%).

Benchmark	Method	Agent	Run 1	Run 2	Run 3	Mean ± std	SR (%)
WebArena	Frozen agent	UI-TARS-7B	68 (26/19/23)	69 (25/21/23)	65 (19/20/26)	67.33 ± 2.08	20.8
	Frozen agent	Qwen3-VL-8B	63 (15/23/25)	64 (23/19/22)	68 (23/23/22)	65.00 ± 2.65	20.1
	AWM-online	UI-TARS-7B	72 (23/28/21)	78 (27/27/24)	71 (19/28/24)	73.67 ± 3.79	22.7
	AWM-online	Qwen3-VL-8B	62 (19/21/22)	55 (19/17/19)	65 (20/23/22)	60.67 ± 5.13	18.7
	DMS	UI-TARS-7B	72 (22/28/22)	71 (25/25/21)	75 (20/27/28)	72.67 ± 2.08	22.4
	DMS	Qwen3-VL-8B	62 (21/20/21)	61 (22/18/21)	64 (23/18/23)	62.33 ± 1.53	19.2
	SOLO (ours)	UI-TARS-7B	89 (25/34/30)	79 (21/30/28)	83 (26/32/25)	83.67 ± 5.03	25.8
	SOLO (ours)	Qwen3-VL-8B	80 (26/27/27)	78 (26/25/27)	84 (26/28/30)	80.67 ± 3.06	24.9
VisualWebArena	Frozen agent	UI-TARS-7B	62 (23/21/18)	73 (24/23/26)	71 (20/25/26)	68.67 ± 5.86	16.7
	Frozen agent	Qwen3-VL-8B	64 (24/20/20)	59 (19/18/22)	60 (22/16/22)	61.00 ± 2.65	14.8
	AWM-online	UI-TARS-7B	63 (25/16/22)	73 (26/22/25)	65 (17/23/25)	67.00 ± 5.29	16.3
	AWM-online	Qwen3-VL-8B	55 (13/20/22)	61 (21/18/22)	65 (21/19/25)	60.33 ± 5.03	14.7
	DMS	UI-TARS-7B	66 (17/26/23)	73 (21/24/28)	70 (19/26/25)	69.67 ± 3.51	17.0
	DMS	Qwen3-VL-8B	64 (22/19/23)	68 (21/21/26)	61 (20/20/21)	64.33 ± 3.51	15.7
	SOLO (ours)	UI-TARS-7B	86 (29/29/28)	81 (29/27/25)	80 (26/28/26)	82.33 ± 3.21	20.0
	SOLO (ours)	Qwen3-VL-8B	82 (22/31/29)	95 (30/32/33)	81 (21/27/33)	86.00 ± 7.81	20.9
MobileWorld	Frozen agent	UI-TARS-7B	6 (2/2/2)	8 (3/2/3)	8 (3/2/3)	7.33 ± 1.15	6.1
	Frozen agent	Qwen3-VL-8B	11 (3/4/4)	13 (4/4/5)	11 (5/4/2)	11.67 ± 1.15	9.7
	AWM-online	UI-TARS-7B	11 (3/5/3)	13 (4/4/5)	11 (3/4/4)	11.67 ± 1.15	9.7
	AWM-online	Qwen3-VL-8B	15 (4/6/5)	13 (4/4/5)	11 (3/4/4)	13.00 ± 2.00	10.8
	DMS	UI-TARS-7B	10 (2/4/4)	9 (4/2/3)	8 (2/3/3)	9.00 ± 1.00	7.5
	DMS	Qwen3-VL-8B	11 (3/3/5)	12 (5/4/3)	12 (4/3/5)	11.67 ± 0.58	9.7
	SOLO (ours)	UI-TARS-7B	12 (4/4/4)	10 (3/4/3)	11 (4/3/4)	11.00 ± 1.00	9.2
	SOLO (ours)	Qwen3-VL-8B	19 (6/7/6)	15 (5/5/5)	14 (5/4/5)	16.00 ± 2.65	13.3

Runs and variance. Every method and variant is run three times on the identical stream in the identical order. Runs differ through the nondeterminism of the environments, of the agents' inference and of the auxiliary models. Every table reports the mean and the sample standard deviation of wins over the three runs, and the success rate of the mean. Fig. 2 is built from the same runs: for each run the success rate over the episodes seen so far, then the mean over the three runs of a cell with a band of one standard deviation across them.

Episode length. Tab. 10 gives the mean number of steps per episode and the share of episodes that end on the step budget, per round, for every cell of Tab. 2 on the web streams and every variant of Tab. 3. The frozen agents and the memory baselines are flat across rounds. With Qwen3-VL-8B the

Table 8: **Every run behind Tab. 3.** Qwen3-VL-8B on both web streams. Wins per run with the round split in parentheses, then mean ± std of wins, success rate (%), and the mean number of updates per run.

Stream	Variant	Run 1	Run 2	Run 3	Mean ± std	SR (%)	Updates
VisualWebArena	Frozen agent	64 (24/20/20)	59 (19/18/22)	60 (22/16/22)	61.00 ± 2.65	14.8	–
	SOLO (full)	82 (22/31/29)	95 (30/32/33)	81 (21/27/33)	86.00 ± 7.81	20.9	196
	w/o window	74 (20/28/26)	80 (23/29/28)	70 (20/25/25)	74.67 ± 5.03	18.2	176
	w/o top-K self-distillation	75 (19/28/28)	83 (21/30/32)	96 (26/37/33)	84.67 ± 10.60	20.6	180
	w/o hindsight relabeling	64 (19/18/27)	71 (24/22/25)	75 (24/25/26)	70.00 ± 5.57	17.0	109
WebArena	Frozen agent	63 (15/23/25)	64 (23/19/22)	68 (23/23/22)	65.00 ± 2.65	20.1	–
	SOLO (full)	80 (26/27/27)	78 (26/25/27)	84 (26/28/30)	80.67 ± 3.06	24.9	177
	w/o window	79 (23/28/28)	73 (22/29/22)	72 (20/29/23)	74.67 ± 3.79	23.0	196
	w/o top-K self-distillation	68 (18/27/23)	77 (22/34/21)	69 (24/26/19)	71.33 ± 4.93	22.0	180
	w/o hindsight relabeling	71 (20/26/25)	69 (20/26/23)	73 (21/27/25)	71.00 ± 2.00	21.9	64

Table 9: **Every run behind Tab. 4.** Wins per run, the judge's precision on that run (share of admitted episodes that the evaluator scores as successes), and the success rate over the runs.

Stream	Auxiliaries	Wins per run	Judge precision per run	SR (%)
VisualWebArena	gpt-5-mini	82 / 95 / 81	0.52 / 0.54 / 0.48	20.9
VisualWebArena	gpt-5	76 / 73 / 84	0.59 / 0.59 / 0.60	18.9
VisualWebArena	gpt-5-nano	80 / 76 / 80	0.46 / 0.41 / 0.42	19.1
VisualWebArena	the agent's own model (Qwen3-VL-8B)	69 / 75 / 69	0.36 / 0.36 / 0.34	17.3
WebArena	gpt-5-mini	80 / 78 / 84	0.63 / 0.67 / 0.67	24.9
WebArena	gpt-5	72 / 76 / 76	0.74 / 0.75 / 0.78	23.0
WebArena	gpt-5-nano	80 / 71 / 81	0.49 / 0.44 / 0.46	23.9
WebArena	the agent's own model (Qwen3-VL-8B)	82 / 77 / 76	0.53 / 0.50 / 0.58	24.2

full method's episodes lengthen over the three rounds, by two to three steps, and the share ending on the budget rises to about one in six. With UI-TARS-7B its episodes shorten slightly and fewer end on the budget than for the frozen agent. The one-hot variant lengthens its episodes far more on both streams. Its updates are also larger: the median gradient norm of a one-hot update is 0.56 to 0.61 in the three runs of each stream, against 0.02 for top-K self-distillation at the same learning rate.

Table 10: **Episode length along the stream.** Mean steps per episode and share of episodes ending on the step budget, per round, mean over the three runs. Every cell of Tab. 2 on the web streams and every variant of Tab. 3.

Stream	Agent	Method	Steps per episode (r1 / r2 / r3)	Budget-ended % (r1 / r2 / r3)
WebArena	UI-TARS-7B	frozen agent	17.0 / 16.5 / 17.5	41 / 40 / 43
		AWM-online	17.4 / 16.4 / 17.5	42 / 36 / 40
		DMS	16.7 / 16.0 / 17.5	41 / 36 / 43
		SOLO	17.8 / 15.7 / 16.1	43 / 34 / 35
	Qwen3-VL-8B	frozen agent	9.1 / 9.5 / 9.1	6 / 8 / 8
		AWM-online	9.0 / 8.1 / 8.0	9 / 7 / 5
		DMS	9.1 / 9.0 / 8.9	10 / 7 / 8
		SOLO	9.4 / 11.0 / 11.9	7 / 12 / 19
		w/o window	9.5 / 10.8 / 11.9	10 / 14 / 17
		w/o top-K self-distillation	9.3 / 17.3 / 25.0	8 / 36 / 74
		w/o hindsight relabeling	9.1 / 9.6 / 9.9	6 / 9 / 10
VisualWebArena	UI-TARS-7B	frozen agent	15.4 / 16.4 / 16.5	40 / 43 / 44
		AWM-online	16.3 / 15.9 / 16.8	42 / 40 / 45
		DMS	16.3 / 16.5 / 15.8	43 / 44 / 41
		SOLO	15.8 / 14.2 / 14.6	41 / 35 / 36
	Qwen3-VL-8B	frozen agent	4.8 / 4.9 / 5.4	2 / 1 / 3
		AWM-online	5.2 / 4.7 / 5.2	2 / 2 / 2
		DMS	5.0 / 4.7 / 4.7	3 / 2 / 1
		SOLO	6.0 / 7.2 / 8.9	4 / 9 / 15
		w/o window	5.2 / 6.2 / 7.9	3 / 5 / 10
		w/o top-K self-distillation	5.2 / 11.0 / 16.3	3 / 19 / 42
		w/o hindsight relabeling	5.2 / 6.0 / 7.9	2 / 6 / 10

E JUDGE AND AUXILIARY MODELS

The judge against the evaluator. Tab. 11 measures the gpt-5-mini judge on SOLO's own episodes in the three runs of each cell: how many episodes it admitted, how many of those the benchmark evaluator scores as successes, and how many evaluator successes there were. On the web streams the judge admits between six and eight of every ten evaluator successes, and between a third and a half of what it admits is not an evaluator success. On MobileWorld it admits nearly every evaluator success, plus about as many episodes that the evaluator rejects with UI-TARS-7B and about half as many with Qwen3-VL-8B.

Table 11: **The gpt-5-mini judge on SOLO's episodes.** Mean over the three runs per cell: judged successes, the true successes among them, the evaluator's successes, precision (true among judged) and recall (judged among true).

Stream	Agent	Judged	True among judged	Evaluator	Precision	Recall
WebArena	UI-TARS-7B	96	58	84	0.60	0.69
WebArena	Qwen3-VL-8B	74	49	81	0.66	0.60
VisualWebArena	UI-TARS-7B	94	57	82	0.61	0.70
VisualWebArena	Qwen3-VL-8B	127	65	86	0.51	0.76
MobileWorld	UI-TARS-7B	23	10	11	0.44	0.94
MobileWorld	Qwen3-VL-8B	25	16	16	0.63	0.98

A perfect judge. Tab. 12 replaces the judge by the benchmark's own evaluator, with gpt-5-mini kept as the proposer and the verifier, so that every admitted episode is a true success. A perfect judge gives the same success rate as gpt-5-mini on both streams, within the run-to-run spread, although gpt-5-mini's admissions are a third to a half false. Removing every false admission therefore does not raise the success rate beyond the run-to-run spread. This is consistent with the soft-filter reading of Sec. 4.4.

Table 12: **A perfect judge against the default judge.** SOLO on Qwen3-VL-8B with the judge replaced by the benchmark evaluator, proposer and verifier unchanged. Wins per run, judge precision, and success rate (%) mean ± std over three runs.

Stream	Judge	Wins per run	Judge precision	Success (%)
VisualWebArena	gpt-5-mini (default)	82 / 95 / 81	0.51	20.9 ± 1.9
VisualWebArena	benchmark evaluator	80 / 78 / 95	1.00	20.5 ± 2.3
WebArena	gpt-5-mini (default)	80 / 78 / 84	0.66	24.9 ± 0.9
WebArena	benchmark evaluator	78 / 71 / 78	1.00	23.4 ± 1.2

The two roles swapped separately. Tab. 13 swaps the two auxiliary roles one at a time on VisualWebArena, the stream on which the agent's own model loses the most in Tab. 4. With the agent's own model as the judge alone, precision falls to about a third and the success rate to 18.2. With it as the proposer and the verifier alone, precision is unchanged but the verified prefixes fall by about half, from about seventy to about thirty-five per run, and the success rate to 17.4. With both roles swapped the success rate is 17.3. Either role alone thus accounts for most of the loss, and the two together add little more.

Table 13: **The auxiliary roles swapped one at a time on VisualWebArena.** SOLO on Qwen3-VL-8B. Wins per run, judge precision and success rate (%) mean ± std over three runs.

Judge	Proposer and verifier	Wins per run	Judge precision	Success (%)
gpt-5-mini	gpt-5-mini	82 / 95 / 81	0.51	20.9 ± 1.9
Qwen3-VL-8B	gpt-5-mini	71 / 70 / 83	0.36	18.2 ± 1.8
gpt-5-mini	Qwen3-VL-8B	76 / 69 / 70	0.48	17.4 ± 0.9
Qwen3-VL-8B	Qwen3-VL-8B	69 / 75 / 69	0.35	17.3 ± 0.8

F COMPUTE AND COST

SOLO adds to each episode one judge call, a proposer and a verifier call on most judged failures, and one gradient step over the window. In the reported runs the auxiliary calls number about two per episode. A proposer call carries every screenshot of the failed episode and runs to five to seven thousand tokens, and a verifier call to about one thousand. On the web streams these additions took between about half a minute and a minute per episode, measured within each run as the time from the end of the rollout to the end of the episode, against rollouts of one to one and a half minutes. Every run occupied one NVIDIA H200 shared with other runs, so absolute times vary with the load at the time of the run and are not compared across runs.

G PROMPTS

Every prompt below is reproduced as used in the reported runs, folded to plain ASCII and re-wrapped to the page width. Braces mark the fields filled at run time.

Agents. UI-TARS-7B runs with its own web prompt, followed by two notes that every method on the web streams shares, one on information tasks and one on step discipline. Qwen3-VL-8B runs with the computer-use tool prompt of its release and the response format of the MobileWorld agent, and the same two notes ride inside the user turn after the instruction. On MobileWorld both agents use the benchmark's own prompts, UI-TARS-7B with the information-task note.

```
You are a GUI agent. You are given a task and your action history,
  with screenshots. You need to perform the next action to complete
  the task.

## Output Format
```
Thought: ...
Action: ...
```

## Action Space
click(start_box='<|box_start|>(x1,y1)<|box_end|>')
type(content='') # If you want to submit your input, use "\n" at the
  end of `content`.
scroll(start_box='<|box_start|>(x1,y1)<|box_end|>',
  end_box='<|box_start|>(x3,y3)<|box_end|>')
press_back() # Go back to the previous page.
press_home() # Go back to the task's starting page.
finished(content='') # Submit the task regardless of whether it
  succeeds or fails.
answer(content='') # Answer user's question.

## Note
- Use English in `Thought` and `Action` part.
- Write a small plan and finally summarize your next action (with its
  target element) in one sentence in `Thought` part.

## User Instruction
{instruction}


## Information tasks
For information-seeking tasks (questions asking how many / how long /
  what / which / to answer with ...), return the requested information
  using answer(content='...') -- do NOT terminate with an empty
  finished(). Only call finished() for non-question tasks.

## Step discipline
Your step budget is limited. Before each action, check whether the
  task goal is already met on the current page -- if it is, emit
```

```
  finished() (or answer(content='...') for questions) immediately
  instead of acting further. If your previous action did not change
  the page, do not repeat it -- choose a different element or a
  different action type.

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>
{"type": "function", "function": {"name": "computer_use",
  "description": "Use a mouse and keyboard to interact with a
  computer, and take screenshots.\n* This is an interface to a web
  page in a browser. The screenshot shows the page content only (no
  address bar or browser buttons); use the `back` and `home` actions
  to navigate between pages.\n* Some pages may take time to load or
  process actions, so you may need to wait and take successive
  screenshots to see the results of your actions.\n* The screen's
  resolution is 1000x1000.\n* Whenever you intend to move the cursor
  to click on an element like an icon, you should consult a screenshot
  to determine the coordinates of the element before moving the
  cursor.\n* If you tried clicking on a program or link but it failed
  to load, even after waiting, try adjusting your cursor position so
  that the tip of the cursor visually falls on the element that you
  want to click.\n* Make sure to click any buttons, links, icons, etc
  with the cursor tip in the center of the element. Don't click boxes
  on their edges.", "parameters": {"properties": {"action":
  {"description": "The action to perform. The available actions
  are:\n* `key`: Performs key down presses on the arguments passed in
  order, then performs key releases in reverse order.\n* `type`: Type
  a string of text on the keyboard.\n* `mouse_move`: Move the cursor
  to a specified (x, y) pixel coordinate on the screen.\n*
  `left_click`: Click the left mouse button at a specified (x, y)
  pixel coordinate on the screen.\n* `left_click_drag`: Click and drag
  the cursor to a specified (x, y) pixel coordinate on the screen.\n*
  `right_click`: Click the right mouse button at a specified (x, y)
  pixel coordinate on the screen.\n* `middle_click`: Click the middle
  mouse button at a specified (x, y) pixel coordinate on the
  screen.\n* `double_click`: Double-click the left mouse button at a
  specified (x, y) pixel coordinate on the screen.\n* `triple_click`:
  Triple-click the left mouse button at a specified (x, y) pixel
  coordinate on the screen (simulated as double-click since it's the
  closest action).\n* `scroll`: Performs a scroll of the mouse scroll
  wheel.\n* `hscroll`: Performs a horizontal scroll (mapped to regular
  scroll).\n* `wait`: Wait specified seconds for the change to
  happen.\n* `terminate`: Terminate the current task and report its
  completion status.\n* `answer`: Answer a question.\n* `back`: Go
  back to the previous page.\n* `home`: Go back to the task's starting
  page.", "enum": ["key", "type", "mouse_move", "left_click",
  "left_click_drag", "right_click", "middle_click", "double_click",
  "triple_click", "scroll", "hscroll", "wait", "terminate", "answer",
  "back", "home"], "type": "string"}, "keys": {"description":
  "Required only by `action=key`.", "type": "array"}, "text":
  {"description": "Required only by `action=type` and
  `action=answer`.", "type": "string"}, "coordinate": {"description":
  "(x, y): The x (pixels from the left edge) and y (pixels from the
  top edge) coordinates to move the mouse to.", "type": "array"},
  "pixels": {"description": "The amount of scrolling to perform.
  Positive values scroll up, negative values scroll down. Required
  only by `action=scroll` and `action=hscroll`.", "type": "number"},
  "time": {"description": "The seconds to wait. Required only by
```

```
    `action=wait`.", "type": "number"}, "status": {"description": "The
    status of the task. Required only by `action=terminate`.", "type":
    "string", "enum": ["success", "failure"]}}, "required": ["action"],
    "type": "object"}}}
</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

Response format for every step:
1) Thought: one concise sentence explaining the next move (no
    multi-step reasoning).
2) Action: a short imperative describing what to do.
3) A single <tool_call>...</tool_call> block containing only the JSON:
    {"name": <function-name>, "arguments": <args-json-object>}.

Rules:
- Output exactly in the order: Thought, Action, <tool_call>.
- Be brief: one sentence for Thought, one for Action.
- Do not output anything else outside those three parts.
- If finishing, use computer_use with action=terminate in the tool
    call.

[user turn, after the current screenshot]
The user query: {instruction}
(For information questions, return the answer with the `answer` action
    (text='...') before terminating.)
Your step budget is limited: if the goal is already met,
    answer/terminate immediately; never repeat an action that did not
    change the page.
Task progress (You have done the following operation on the current
    page): {steps}

You are a GUI agent. You are given a task and your action history,
    with screenshots. You need to perform the next action to complete
    the task.

## Output Format
```
Thought: ...
Action: ...
```

## Action Space
click(start_box='<|box_start|>(x1,y1)<|box_end|>')
long_press(start_box='<|box_start|>(x1,y1)<|box_end|>', time='')
type(content='') # If you want to submit your input, use "\n" at the
    end of `content`.
scroll(start_box='<|box_start|>(x1,y1)<|box_end|>',
    end_box='<|box_start|>(x3,y3)<|box_end|>')
press_home()
press_back()
open_app(content='') # Open an app specified by `content`.
finished(content='') # Submit the task regardless of whether it
    succeeds or fails.
answer(content='') # Answer user's question.

## Note
- Use English in `Thought` and `Action` part.
- Write a small plan and finally summarize your next action (with its
```

```
  target element) in one sentence in `Thought` part.

## User Instruction
{instruction}


## Information tasks
For information-seeking tasks (questions asking how many / how long /
  what / which / to answer with ...), return the requested information
  using answer(content='...') -- do NOT terminate with an empty
  finished(). Only call finished() for non-question tasks.

# 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>
{"type": "function", "function": {"name": "mobile_use", "description":
  "Use a touchscreen to interact with a mobile device, and take
  screenshots.\n* This is an interface to a mobile device with
  touchscreen. You can perform actions like clicking, typing, swiping,
  etc.\n* Some applications may take time to start or process actions,
  so you may need to wait and take successive screenshots to see the
  results of your actions.\n* The screen's resolution is 999x999.\n*
  Make sure to click any buttons, links, icons, etc with the cursor
  tip in the center of the element. Don't click boxes on their edges
  unless asked.", "parameters": {"properties": {"action":
  {"description": "The action to perform. The available actions
  are:\n* `click`: Click the point on the screen with coordinate (x,
  y).\n* `long_press`: Press the point on the screen with coordinate
  (x, y) for specified seconds.\n* `swipe`: Swipe from the starting
  point with coordinate (x, y) to the end point with coordinates2 (x2,
  y2).\n* `type`: Input the specified text into the activated input
  box.\n* `answer`: Output the answer.\n* `system_button`: Press the
  system button.\n* `open`: Open an app on the device.\n* `wait`: Wait
  specified seconds for the change to happen.\n* `terminate`:
  Terminate the current task and report its completion status.",
  "enum": ["click", "long_press", "swipe", "type", "answer",
  "system_button", "open", "wait", "terminate"], "type": "string"},
  "coordinate": {"description": "(x, y): The x (pixels from the left
  edge) and y (pixels from the top edge) coordinates to move the mouse
  to. Required only by `action=click`, `action=long_press`, and
  `action=swipe`.", "type": "array"}, "coordinate2": {"description":
  "(x, y): The x (pixels from the left edge) and y (pixels from the
  top edge) coordinates to move the mouse to. Required only by
  `action=swipe`.", "type": "array"}, "text": {"description":
  "Required only by `action=type`, `action=answer`, and
  `action=open`.", "type": "string"}, "time": {"description": "The
  seconds to wait. Required only by `action=long_press` and
  `action=wait`.", "type": "number"}, "button": {"description": "Back
  means returning to the previous interface, Home means returning to
  the desktop, Menu means opening the application background menu, and
  Enter means pressing the enter. Required only by
  `action=system_button`", "enum": ["Back", "Home", "Menu", "Enter"],
  "type": "string"}, "status": {"description": "The status of the
  task. Required only by `action=terminate`.", "type": "string",
  "enum": ["success", "failure"]}}, "required": ["action"], "type":
  "object"}}}
</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

Response format for every step:
1) Thought: one concise sentence explaining the next move (no
  multi-step reasoning).
2) Action: a short imperative describing what to do.
3) A single <tool_call>...</tool_call> block containing only the JSON:
  {"name": <function-name>, "arguments": <args-json-object>}.

Rules:
- Output exactly in the order: Thought, Action, <tool_call>.
- Be brief: one sentence for Thought, one for Action.
- Do not output anything else outside those three parts.
- If finishing, use mobile_use with action=terminate in the tool call.

[user turn, after the current screenshot]
The user query: {instruction}
Task progress (You have done the following operation on the current
  device): {steps}
```

Judge. The judge receives the system prompt below, an evaluator prompt followed by a block of verification rules, and a user message that gives the task, every action of the episode in order with the screenshot after up to 48 uniformly sampled steps, and the final state, in the format shown after it.

```
You are an expert evaluator for mobile agent task completion. Your
  role is to determine whether a given task has been successfully
  completed based on the provided trajectory evidence.

## Evaluation Guidelines:

1. **Outcome-Focused**: Judge based on whether the primary objective
  was achieved, not the path taken.

2. **Use All Provided Evidence**: Base your judgment on all available
  information -- screenshots, actions, agent reasoning, and UI element
  lists. Do not assume information beyond what is provided.

3. **Mid-Trajectory Success**: The task may be completed in an
  intermediate step rather than the final one. Additional actions
  after completion do not invalidate success, unless they explicitly
  undo it.

4. **Corrective Actions**: If the agent made mistakes but corrected
  them and achieved the goal, that counts as success.

5. **UI State Indicators**: Pay attention to visual indicators like
  selected tabs, checkboxes, highlighted items, and confirmation
  messages.

Be balanced in your judgment - avoid being overly strict (missing true
  successes) or overly lenient (accepting failures).

VERIFICATION REQUIREMENTS (v1.1). Apply each rule only when the task
  involves
its situation; otherwise judge under the existing guidelines
  unchanged. When a
rule applies and its required evidence is absent or ambiguous, mark
  failure.

R1 -- State commitment. When the task requires creating, changing, or
submitting persistent state (saving an edit, placing an order,
  posting,
```

```
updating a status or setting), success requires evidence that the
  committing
action occurred (e.g., Save / Submit / Place Order / Post clicked)
  and, where
visible, the post-commit state (confirmation message, updated value
  shown
outside the editor, order-confirmation page). Entered-but-uncommitted
  input --
text typed into a field, items sitting in a cart, an edit visible only
  inside
an editor -- is not completion.

R2 -- Empty results. If the final answer or state amounts to "none /
  no
results / empty," verify from the evidence that the query or filter
  was
correctly constructed: correct field, correct spelling and wording of
  the
value, correct date range or option actually applied and visible. An
  empty
result produced by a malformed, misspelled, mis-scoped, or wrong-field
  query
is a failure even though the page truthfully shows no results. If
  query
correctness cannot be verified from the evidence, do not accept the
  empty
result as success.

R3 -- Selection constraints. When the task specifies which item among
candidates -- a superlative or ordering (latest, newest, first,
  cheapest,
most X) or a uniquely identifying property -- verify the selected item
satisfies that constraint with visible evidence (sort order applied,
  the
relevant dates/prices/attributes shown). Membership in the right set
  is not
sufficient. When the task prescribes a specific method or path (e.g.,
  browse
a stated category, use a stated page or tool), the prescribed path
  must have
been used; an alternative route is a failure even if it reached a
similar-looking result.

R4 -- Aggregate completeness. When the task asks for an aggregate or
  extremum
over a collection (a range, maximum, minimum, count, total), verify
  the
evidence covers the full collection -- pagination exhausted or a
  total/count
indicator visible -- not only the first page or a partial view.
  Answers
extrapolated from a visibly partial collection are failures.

## Task
{task}

## Agent Trajectory
The following shows the agent's execution trajectory in chronological
  order. Each step's screenshot is placed immediately after its
  description.

### Step {step_num}{step_label}
**Action**: {action}
[Screenshot attached]
```

```
### Step {step_num}{step_label}
**Action**: {action}

### Step {step_num} (Final State)
[Screenshot attached]
{optional_ui}

## Your Judgment
Based on the trajectory above, determine if the task was successfully
  completed.

Reply in the following format:
Result: <1 for success, 0 for failure>
Confidence: <high/medium/low>
Reason: <brief explanation in 1-2 sentences>
```

Proposer. The proposer receives the system prompt below and a user message with the original instruction and, for every step, the agent's thought, its action and the screenshot it saw, followed by the screenshot after the last action.

```
You are auditing a web-agent trajectory that FAILED its task. You will
  see the ORIGINAL task and, for each
step, the screenshot the agent saw BEFORE acting, its thought and its
  action (click(start_box='(x,y)') with x,y
in 0-1000 relative screen coordinates; type(content=...); scroll(...);
  press_back(); finished(...); answer(...)).

Your job is hindsight relabeling: decide whether some PREFIX of this
  trajectory correctly completed a coherent
task of its own -- a SUBTASK or PREREQUISITE of the original task
  (e.g. "open the issues page of repository X",
"search the store for 'wireless mouse' and open the results",
  "navigate to the settings page of forum F") --
even though the original task was not completed.

Rules for a valid proposal:
1. The pseudo task must be a subtask, prerequisite or strictly
  narrower version of the ORIGINAL task, on the same
   site and about the same objects; never an unrelated task, and never
     the original task itself.
2. It must be COMPLETED by the agent's actions up to some step k
  (0-based): the screenshot the agent saw at step
   k+1 (i.e. after acting at step k) must show the task's result.
     Choose the EARLIEST such k; do not include
   later steps that wander, loop or undo progress.
3. It must be non-trivial: at least one navigation or typing action
  beyond loading the start page; "scroll the
   page", "open the homepage" or "click around" are not tasks.
4. It must be stated like a user instruction (one sentence,
  imperative, concrete names/values), and be
   verifiable from the final screenshot alone.
5. If no prefix satisfies all of the above, answer valid=false.
6. The pseudo task must END ON A RESULTING PAGE OR STATE produced by a
  page-changing action (a search
   results page, a product / repository / report page, a submitted
     form, a changed setting). Text typed
   into a field that was not submitted, an opened menu, or a
     highlighted element is NOT a completed task.
7. The pseudo task must describe a SUCCESSFUL outcome that a user
  would ask for. An error message, a
   validation failure, an empty or 'no results' page, or an attempt
     that did not go through is NOT a
   completed task.

Respond with ONLY a JSON object:
```

```
{"valid": true|false, "pseudo_task": "<instruction or empty>",
  "end_step": <k or -1>,
 "relation": "subtask"|"prerequisite"|"narrower"|"none", "reason":
   "<one sentence>"}

ORIGINAL TASK (failed): {instruction}

The trajectory has {n} steps.
--- Step 0 (screenshot before acting below)
Thought: {thought 0}
{action 0}
[screenshot 0]
--- Step 1 (screenshot before acting below)
...
--- Screenshot after the last action (step {n-1}):
[screenshot n]
```

Verifier. The verifier receives only the proposed instruction, the actions of the prefix and the screenshot after its last step.

```
You are a strict verifier for a web-agent task. You will see a TASK,
  the list of actions the agent took, and
the FINAL screenshot after those actions. Decide whether the final
  screenshot shows that the task has been
completed as stated (correct page, correct object, requested state
  reached). Be conservative: if the
screenshot does not by itself demonstrate completion, or the task is
  vague or trivial, answer false.

Respond with ONLY a JSON object: {"verified": true|false,
  "confidence": <0-1>, "reason": "<one sentence>"}

[user turn]
TASK: {pseudo task}

ACTIONS TAKEN:
0: Action: {action 0}
1: Action: {action 1}
...

FINAL SCREENSHOT:
[screenshot after step k]
```

Memory baselines. AWM-online induces its workflows with the authors' instruction, reproduced first, and a one-shot example rewritten in our agents' action vocabulary, reproduced in its UI-TARS-7B form. The induced workflow file enters the agent's prompt as the block shown last, with the action-space note appended to the file.

```
Given a list of web navigation tasks, your task is to extract the
  common workflows to solve these tasks.
Each given task contains a natural language instruction, and a series
  of actions to solve the task. You need to find the repetitive subset
  of actions across multiple tasks, and extract each of them out as a
  workflow.
Each workflow should be a commonly-reused sub-routine of the tasks. Do
  not generate similar or overlapping workflows. Each workflow should
  have at least two steps. Represent the non-fixed elements (input
  text, button strings) with descriptive variable names as shown in
  the example.
Keep the values of invariant elements, e.g., id of "Search" or
  "Customers", as they will share and stay invariant across tasks.
Try to generate as many workflows that can cover all the tasks in the
  input list.

## Concrete Examples
```

```
Query: What is the date of my first purchase on this store?
Actions:
<think>
To find the first purchase I need the order history, which lives under
  the account menu.
</think>
<action>
Click on the account menu in the top-right corner of the page.
</action>

<think>
The order history is the "My Orders" entry in the account page's left
  sidebar.
</think>
<action>
Click the "My Orders" entry in the left sidebar.
</action>

## Summary Workflows

Workflow 1: Open the account's order history
<think>
To reach anything about past orders I first open the account menu in
  the top-right corner.
</think>
<action>
Click on the account menu in the top-right corner of the page.
</action>

<think>
From the account page the order history is the "My Orders" entry in
  the left sidebar.
</think>
<action>
Click the "My Orders" entry in the left sidebar.
</action>

Workflow 2: Search the store for a product
<think>
I put the product I am looking for, {product-name}, into the site's
  search box.
</think>
<action>
Type {product-name} into the search box and press Enter.
</action>

## Workflows
These sub-routines were induced from earlier tasks on this website
  that were completed successfully. Reuse the matching one,
  substituting this task's own values for the {{placeholders}} -- read
  those from the current instruction and the current screen, never
  from a workflow.
{workflows}

[action-space note appended to every induced workflow file, UI-TARS
  form]

(Write each step as the action you would take; keep the page's own
  button and field names, and substitute this task's values for every
  {placeholder}.)

[Qwen3-VL form]

(Each step is a computer_use tool call, exactly as you emit them;
```

```
  substitute this task's values for every {placeholder}.)
```

Darwinian Memory shows the agent its retrieved units in the block below.

```
## Retrieved memory
Each unit below was recorded when an earlier task with a similar goal
  succeeded from a similar starting state. Replay the matching steps
  where they still fit what you see; this task's own values must come
  from the current instruction and screen.
{units}

[one unit]
### Goal: {goal}
Precondition: {precond}
Actions:
{steps}
```