WEBACT: TEST-TIME LEARNING OF VERIFIABLE ACTION INTERFACES FOR WEB AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

A web agent that has just finished a routine cannot call that routine again. Its next task begins again from browser primitives, and the agent works out the same clicks and keystrokes a second time. Making the routine callable via skill induction solves only part of the problem. A routine that worked once can be invoked on a page where it cannot work, and even on the right page it can run to the end without producing the effect that the task needed. We study test-time learning of verifiable action interfaces, where an agent learns reusable operations together with the conditions under which they should be used. Our framework, WEBACT, learns an *ActSpec* for each routine: a parameterized plan, a rule for finding its targets on the current page, a condition for when it may be invoked, and a check on the effect it should produce. Every one of these fields is taken from the trace that the routine was learned from. Successful attempts add operations, failed attempts add constraints, and each invocation outcome adjusts which operations the planner prefers, all without changing model weights. On WebArena tasks, the learned action interface improves task success by 32.8% relative to a primitive-action baseline while using 42.9% fewer tokens per task, and the gain holds on live websites. Removing either of the two learned conditions lowers success. Further trace analysis revealed the consequences of applying a wrong skill: once the planner selects an operation on the wrong page, it adopts a sub-goal that the task never had, and rejecting the call does not remove that sub-goal.

1 INTRODUCTION

An agent asked to find a product on a shopping site finds the search box, types a query, submits it, waits, and reads the results. Ten minutes later it is asked for a different product, and it performs the same five steps again, because the routine it just finished was not saved in any form the next task can call. This is the cost of a fixed action interface: the agent can only plan in browser primitives, so a routine it has already performed must be planned again from single clicks and keystrokes.

Storing the routine as a skill so that it can be called again is the obvious fix, and it is not enough. A search routine learned on a shopping page may be selected on a forum page where it cannot work, and the planner then pursues a sub-goal that the task never had. Even on the correct page, every click and keystroke can succeed while the effect that the task asked for never appears. An agent therefore has to learn two things at once: the operation, and the conditions under which it should be used. We call this problem *test-time learning of verifiable action interfaces*.

Prior work offers several ways to turn experience into reusable capabilities, and each already cares about reliability. Workflow memory stores procedures, ASI induces and validates programmatic skills, SkillWeaver discovers website APIs and filters them by pre-conditions, WALT validates tool scripts, and ContractSkill uses executable contracts to locate and repair failures inside a skill (Wang et al., 2025b;a; Zheng et al., 2025; Prabhu et al., 2026; Lu et al., 2026). What is still open is how a stream of experience should change the interface as a whole. We propose WEBACT, in which successful traces supply new operations, failed attempts supply constraints, and the outcome of each invocation updates which operations the planner prefers. Experience thus shapes both what the agent can do and when it chooses to do it.

We write a reusable routine as an *ActSpec*. An ActSpec holds a parameterized plan, a rule for finding its targets on the current page, a condition for when it may be invoked, and a check on the effect it

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.

should produce. Consider a search operation: it needs a query argument, an available search box, and evidence that the results match the query that it was given. Every one of these fields is taken from the trace that the operation was learned from. The typed text becomes the argument, the page state on entry becomes the invocation condition, and the page state on exit becomes the effect check. WEBACT learns these operations without changing any model weights. At each step it keeps only the operations whose condition holds, ranks them by how well they have worked before, resolves their targets, runs the plan, and checks the result. Browser primitives remain available whenever no learned operation applies.

We evaluate on two benchmarks. On WebArena, WEBACT learns an interface from a warm-up split, freezes it, and is then evaluated on 613 held-out tasks. Task success rises from 20.39% to 27.08%, a 32.8% relative gain, with 18.7% fewer planner calls and 42.9% fewer tokens per task. The gain does not depend on which tasks the interface was learned from: five different warm-up sets all beat the baseline. On the live websites of WebVoyager, we give WEBACT the same exploration traces that SkillWeaver uses, and build six operations from them. Those six operations solve 35.7% of the tasks. The skills that SkillWeaver builds from the same traces solve 20.2%, and WALT solves 27.9%. If we let WEBACT keep learning during evaluation, its result rises to 44.2%.

We also conduct an ablation study. Removing the invocation condition lets the planner pick learned operations on pages where they do not apply, and success drops to zero: the agent commits to the wrong sub-goal, fails, and stops early, even though browser primitives were available the whole time. Two further settings keep the invocation condition but weaken target finding, or freeze the preference updates. In both, every learned call is rejected, and yet the agent solves 28 and 29 of 85 tasks. Acceptance and task success are therefore not the same measurement, and the traces show why. A call rejected at the final step was at least invoked on a page where the operation applied, so the agent is working on a sub-goal the task does require and can finish it with browser primitives. A call rejected because the operation did not apply at all leaves the agent working on a sub-goal the task never contained.

Our contributions are:

- **Learning operations together with their conditions of use.** WEBACT derives parameterized operations, invocation conditions, and effect checks from interaction traces, and feeds failure-derived constraints and invocation outcomes back into later reuse, without updating model weights.
- **The learned conditions matter for task success.** Removing the invocation condition drops Reddit success from 37/85 to 0/85, because learned operations then get applied on the wrong pages. Removing the effect check drops it to 24/85, because calls are accepted that never achieved what the task needed.
- **Learned routines transfer within an application.** A frozen interface gains 32.8% relative success at 42.9% fewer tokens on 613 held-out WebArena tasks, and the gain remains when we change the source tasks and the task templates. On live websites it solves more tasks than the skills SkillWeaver builds from the same traces.

2 LEARNING AN ACTION INTERFACE

We study a web agent that learns executable operations from a stream of tasks on one application, starting from browser primitives such as click, type, scroll, wait, and extract, with the backbone model held fixed throughout.

Interaction traces. For an attempt of length T, the agent records

$$\tau = \big(s_0, \{(a_t, \mathtt{tgt}_t, \Delta_t)\}_{t=1}^{T}, y\big), \tag{1}$$

where s_0 is the initial observable state, a_t is a primitive action with arguments, $\mathtt{tgt}_t$ describes its target, and Δ_t records observable changes such as route transitions, region refreshes, and confirmation messages. The evaluation environment supplies a task outcome $y \in \{0, 1\}$. Feedback about an entire attempt is thus separated from evidence about a local interaction.

Executable experience. Let $\mathcal{A}_{\mathrm{prim}}$ denote the primitive actions and $\mathcal{A}_{\mathrm{task}}$ the learned library. The planner acts in their union, subject to learned constraints $\mathcal{C}$. Each operation in the library takes the following form.

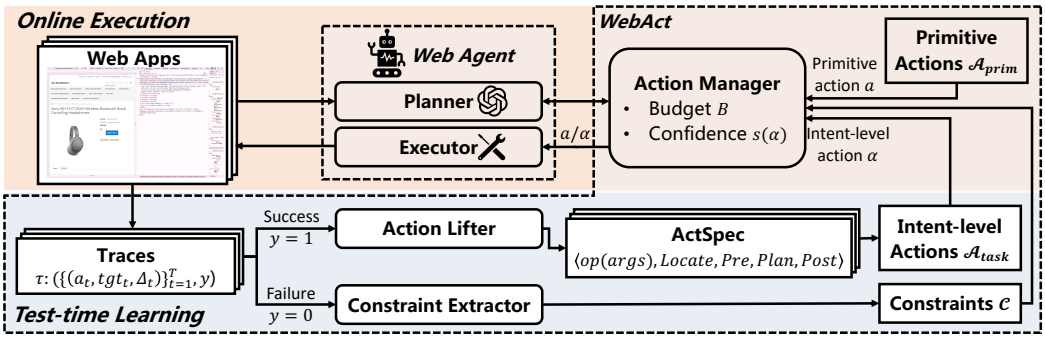

Figure 1: WEBACT learns an action library from successful traces and pruning constraints from failures. Runtime checks govern invocation and provide evidence for confidence updates.

Definition 2.1 (ActSpec)**.** An ActSpec is a parameterized action contract

$$\alpha = \langle \texttt{op(args)}, \texttt{Locate}, \texttt{Pre}, \texttt{Plan}, \texttt{Post} \rangle.$$

For arguments x of the declared types, $\texttt{Pre}_\alpha(s, x) \in \{0, 1\}$ tests applicability in observable state s. The locator $\texttt{Locate}_\alpha(s, x)$ returns a binding b of targets to current-page elements, or failure $\perp$. The plan $\texttt{Plan}_\alpha(x, b)$ is a finite sequence of browser primitives. The outcome predicate $\texttt{Post}_\alpha(s, s', x) \in \{0, 1\}$ tests the specified observable changes between entry state s and resulting state s'.

Writing the conditions into the contract means the agent can check them by looking at the page, rather than assuming them. A search contract, for example, takes a new query, finds the search input on the current page, runs the routine, and then checks that the page shows a result state. Whether the task as a whole succeeds still depends on what the planner does with that result.

Acquisition and reuse. In frozen evaluation, the agent learns its interface before held-out tasks begin and then stops induction; in online evaluation, completed tasks continue to supply new actions and constraints. We report the cost of reuse separately from the cost of acquisition. One learned call can execute several browser primitives, so a count of planner calls and a count of browser events measure different things.

3 WEBACT

3.1 SUCCESS EXPANDS THE INTERFACE; FAILURE CONSTRAINS IT

After each attempt, WebAct records the interaction trace and task outcome (Figure 1). Successful traces supply segments for candidate ActSpecs; unsuccessful traces supply context-specific patterns for constraint induction. Invocation feedback updates preferences among the available operations. Subsequent tasks use the resulting operations and constraints through the same planner.

Algorithm 1 states the learning loop. Initial induction starts from an empty interface; online learning can continue from an existing one. Frozen evaluation stops these induction updates while continuing to use the learned operations and constraints.

3.2 ACTSPEC MAKES ROUTINES CALLABLE, FINDABLE, AND CHECKABLE

A reusable routine has to tolerate two kinds of change: the request differs from the one it was learned on, and so does the page. These give three requirements, and each requirement corresponds to one part of the contract (Figure 2).

First, because the request changes, the routine must be *callable with new arguments*. A typed signature separates the values that belong to a particular task from the routine itself. Strings, enumerations, numbers, and structured arguments cover search, filtering, and form filling, and the plan binds each

Algorithm 1 Interface learning from a task stream.

Require: Task stream $\{\mathcal{T}_i\}$, primitive actions $\mathcal{A}_{\text{prim}}$, per-attempt execution cap H, initial interface $(\mathcal{A}_0, \mathcal{C}_0)$

$\mathcal{A}_{\text{task}} \leftarrow \mathcal{A}_0$; $\mathcal{C} \leftarrow \mathcal{C}_0$
for each task $\mathcal{T}_i$ in the learning stream **do**
    Execute the attempt protocol with cap H, using applicable actions in $\mathcal{A}_{\text{prim}} \cup \mathcal{A}_{\text{task}}$ and constraints $\mathcal{C}$
    Record primitive traces and task outcomes (τ, y)
    if $y = 1$ **then**
        $\mathcal{A}_{\text{task}} \leftarrow \mathcal{A}_{\text{task}} \cup \textsc{LiftSuccess}(\tau)$
    else
        $\mathcal{C} \leftarrow \mathcal{C} \cup \textsc{ExtractConstraints}(\tau)$
return $\mathcal{A}_{\text{task}}, \mathcal{C}$

Field	Illustrative search contract
Signature	`Search(query: string)` exposes a new query as an argument.
Pre	Require a visible, enabled search input in the expected page context.
Locate	Resolve that input by role/label, text anchor, or container; reject unresolved ambiguity.
Plan	Bind `query` to the input, submit, and wait for the result state.
Post	Check a refreshed result region or a URL/header reflecting the query.

Figure 2: Illustrative search contract. New query arguments reuse the same routine. An unavailable input blocks invocation; missing outcome evidence fails verification. Task completion requires the planner to use the result correctly.

argument to a primitive operation (Cai et al., 2024; Wang et al., 2024b). Second, because the page changes, the targets must be *findable at invocation time*. An element can move, change its identifier, or sit next to other elements that look like it (Deng et al., 2023; Zheng et al., 2024), so `Locate` searches for it by accessibility role and label, by nearby visible text, and by which form, panel, or modal contains it, using descriptions taken from the source trace. If more than one element matches, `Locate` reports failure and the planner continues with primitives. Third, a page can look right without the operation being applicable, and an operation can run without succeeding, so both must be *checkable in context*. `Pre`, the invocation condition, tests the route, visibility, and readiness of the page before the call. `Post`, the effect check, looks for a confirmation, a route change, or a refreshed result region afterwards. For pages that load asynchronously, the plan can include waits and intermediate checks.

3.3 Successful Traces Become Parameterized Operations

WebAct segments a successful trajectory at observable state transitions: route changes, modal transitions, and region refreshes. These boundaries identify candidate interactions within a larger task, such as searching and then applying a filter. Every field of the resulting contract is then taken from that same segment. Take the search example. The entered text becomes a query argument bound to a text-entry step in the plan, and the input's role and label supply its locator. The page state on entry supplies the condition that a search input is available, and a changed route or result region on exit supplies the candidate effect check. A language model supplies the operation's semantic description. No part of the contract is therefore written independently of the interaction that produced it. One consequence is that if the source observation was uninformative, the resulting check is also uninformative. Section 4.5 presents a recorded case in which the plan generalizes to new queries but the effect check does not. Structural validation checks consistency between argument definitions, placeholders, and plan bindings, with automatic repair for detected inconsistencies. Algorithm 2 summarizes induction; Appendix B details the checks and structural validation.

3.4 Failed Attempts Become Reusable Constraints

A failed attempt tells the agent where the current interface should behave differently. WebAct extracts two patterns from unsuccessful traces: repeated actions after a page transition without a corresponding state change indicate a readiness problem, and repeated selections among similar target descriptors without the expected effect indicate a disambiguation problem. A constraint $c = \langle \phi(s), \pi(a, \texttt{tgt}) \rangle$ pairs an observable context ϕ with an action/target pattern π to suppress: a readiness constraint requires stabilization before another attempt, and a disambiguation constraint suppresses a repeatedly unsuccessful descriptor in the same context. The planner uses these constraints to filter candidate actions, and the executor can use them to strengthen applicability checks. Failed attempts trigger persistent constraint induction, while individual invocation outcomes supply feedback about the corresponding action (Appendix A).

3.5 Invocation Feedback Guides Future Reuse

At each decision, WebAct filters the learned library by applicability and active constraints, then prefers actions with higher confidence; primitives remain available when no learned routine applies. The executor resolves targets, runs the parameterized plan, and checks the resulting state, returning control to primitive planning when execution fails. Each new ActSpec starts with $s(\alpha) = 1$, and recorded executor outcomes update its confidence:

$$s(\alpha) \leftarrow \begin{cases} \min(s_{\max}, s(\alpha) + 1), & \text{executor succeeds,} \\ \max(0, s(\alpha) - \lambda), & \text{executor fails.} \end{cases}$$

We use $s_{\max} = 10$ and $\lambda = 2$, processing outcomes after each task. Confidence ranks applicable candidates, while accumulated usage and failure statistics separately govern retirement, so past reliability affects preference only after the current page satisfies applicability checks. Selection, effect checking, and preference therefore act at three different points in a reuse decision. When a call fails and the planner falls back to primitives, it continues from the current page and the history it has already built up, not from a cleared state. Section 4.5 measures what each of these controls contributes.

4 Evaluation

The evaluation asks four questions in turn: how much an interface learned in advance helps on held-out tasks, whether that help depends on which tasks it was learned from, whether it still holds on live websites, and what happens when we remove each runtime condition.

4.1 Experimental Setup

Benchmarks and splits. We evaluate five WebArena applications: GitLab, Map, Reddit, Shopping, and Shopping-admin (Zhou et al., 2024b). Within each we randomly assign about 20% of tasks to warm-up and hold out the rest, giving 613 evaluation tasks; WebAct learns during warm-up and stops before evaluation. The two sets share no tasks, and 605 of the 613 evaluation tasks use templates warm-up never contained (Appendix D). For live websites we use the 129-task, definite-answer WebVoyager subset and task order following AgentOccam, which allows automatic answer checking (He et al., 2024; Yang et al., 2025). We keep access failures and anti-bot interruptions in the evaluation, and use the pages without WebArena-specific cleaning.

Methods, protocols, and metrics. AgentOccam supplies the primitive-action baseline, and WALT and SkillWeaver supply further WebVoyager comparisons with their vision or screenshot inputs enabled (Prabhu et al., 2026; Zheng et al., 2025). We use Gemini-2.5-Flash with a budget of 15 planner decisions per attempt, and WebArena allows at most three attempts per task. One learned-action invocation uses a single planner decision while executing several browser operations, and comparisons within an experiment share tasks and retry rules. A frozen interface stops acquiring operations and constraints but keeps checking invocations, updating preferences, and retiring operations; the online configuration keeps acquiring as well (Appendix H). We report task success, the planner calls and tokens spent per evaluation task, and executor acceptance, the fraction of learned-action invocations

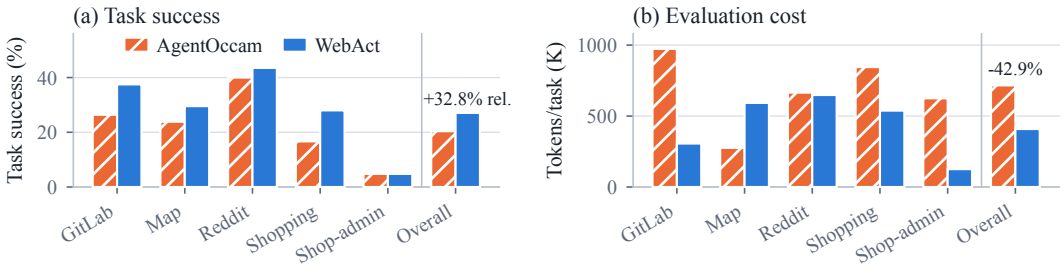

Figure 3: Frozen-interface evaluation on 613 held-out WebArena tasks. The learned interface raises success on four of five applications and lowers cost on four; Map is the exception on cost and Shopping-admin on success. Table 3 lists every value, including action calls and executor acceptance.

the executor accepted because the targets resolved and the effect check passed. Accepting a call says nothing about whether the task was completed, so we report the two separately throughout. For WEBACT on WebVoyager the call counts are primitive decisions only, with learned-action invocations listed separately, and costs reported for a frozen interface cover reuse and exclude the warm-up that produced it.

4.2 LEARNED INTERFACES IMPROVE TASK COMPLETION

WEBACT raises WebArena success from 125/613 (20.39%) to 166/613 (27.08%), a 32.8% relative improvement, while reducing action calls by 18.7% (10.99 to 8.93) and tokens by 42.9% (716.1K to 408.7K) per evaluation task (Figure 3). The agent stops learning before these tasks begin, so the gains come from reusing an interface it already has. Success rises on four applications, is unchanged on Shopping-admin, and 149/175 invocations are accepted.

The gain is largest where routines are long. GitLab and Shopping improve most, from 38 to 54 and from 25 to 42 successes. Both contain search, navigation, and form interactions that recur across tasks, and each of those interactions takes several primitive decisions to perform. On Shopping-admin the benefit appears in cost rather than success: success stays at 7/146, while calls fall from 12.68 to 7.54 and tokens from 625.1K to 125.9K. Map is the exception. Its success improves from 23.86% to 29.55%, but its tokens rise from 275.6K to 592.5K. A likely reason is that Map's primitives, such as zooming and panning, already express much of what its tasks ask for, so an operation has little left to compress. Among 23 inspected cases where WEBACT succeeds and AgentOccam fails, ten baseline runs exhaust their step budget and three loop in repeated recovery (Appendix C).

Comparing our baseline with its published numbers. Our AgentOccam results are lower than the ones reported in its original paper. That paper evaluates all tasks with GPT-4-Turbo and a 20-step budget (Yang et al., 2025), whereas we evaluate the 80% held-out split with Gemini-2.5-Flash and 15 decisions, applying both settings to both methods. A separate configuration with a stronger backbone shows how the comparison behaves when the baseline is near its published strength. On GitLab with Gemini-3.5-Flash-Lite, AgentOccam reaches 56/144 (38.89%), close to the 37.8% its authors report for that application, while WEBACT reaches 59/144 (40.97%) using 7.89 versus 9.15 calls per task. This configuration differs from our main experiment in its task split and its 23-operator library as well as its backbone, so we cannot say how much of the difference from the published rate comes from each of the three (Appendix F). What it does show is that the interface still completes more tasks at lower cost against a stronger baseline, although by a smaller margin.

4.3 REUSE EXTENDS BEYOND THE WARM-UP TASKS

An interface would not be worth learning if it only helped on tasks that resemble the ones it was learned from. We therefore vary the warm-up set in three ways: which tasks it contains, how many tasks it contains, and whether the evaluation tasks come from the same WebArena task templates.

Different source sets retain the gain. Of 106 Reddit tasks we divide 105 into five disjoint 21-task warm-up sets, always holding one task out. Each set produces a frozen library, evaluated on the other 85 tasks, with AgentOccam run on those same tasks. WEBACT improves task success and both cost measures in all five folds: mean success is $44.94\% \pm 0.98\%$ versus $41.41\% \pm 1.93\%$, with calls falling from 9.60 to 7.95 and tokens from 592K to 472K, the deviations describing variation across five partitions whose evaluation sets overlap. We also raise the warm-up share from 5% to 20% with the evaluation set fixed. Acceptance rises from 10/13 to 14/16 while calls and tokens decline, and even the smallest set beats the 40.00% baseline, though it spends more tokens to do so. Task benefit can therefore appear before the interface becomes cheaper (Figure 5 and Appendix D).

Reuse also crosses task templates. In the main evaluation 605 of 613 tasks come from templates warm-up never contained, and for Shopping and Reddit the two sets share no template at all, yet success improves from 25 to 42 and 34 to 37 (Appendix D). The complete goals differ while the interactions inside them, such as search-and-open, filtering, and navigation, recur, and that is the level at which an operation transfers.

4.4 REUSE TRANSFERS TO LIVE WEBSITES

WebArena pages are cleaner and more regular than pages on the open web, so we repeat the test on live sites. We use three configurations: a frozen library built from another method's exploration traces, a held-out split of the tasks, and a setting in which the agent keeps learning as it works through the stream.

Table 1: WebVoyager definite-answer subset (129 tasks). WEBACT calls count primitive decisions. The final row evaluates online induction separately from frozen reuse; each system uses its own runtime cost accounting.

Method	Protocol	Success	Calls/task	Tokens (K)
AgentOccam	None	41/129 (31.8%)	7.93	604.4
WALT	Frozen	36/129 (27.9%)	7.06	53.4
SkillWeaver	Frozen	26/129 (20.2%)	7.11	50.9
WEBACT	Frozen	**46/129 (35.7%)**	5.89	577.8
WEBACT	Online	57/129 (44.2%)	5.73	364.3

From the same exploration data, our interface completes more tasks. SkillWeaver explores a site, collects traces, and turns them into skills. We take those same traces and build six ActSpecs from them, so the two systems start from identical experience and differ only in what they build out of it. The six operations solve 46/129 tasks, while SkillWeaver's own skills solve 26/129 and WALT, which collects its own warm-up data, solves 36/129 (Table 1). Holding the traces fixed rules out one explanation, that WEBACT had more exploration available. It does not show that the representation alone is responsible, since the systems also differ in how they build, retrieve, and execute what they learn. Most of the reuse comes from two routines: of the 88 accepted invocations, 40 are a calculation routine and 39 are a search routine, so a library of six operations can serve many tasks across several sites. WALT and SkillWeaver report fewer tokens than WEBACT, so they trade success for cost differently.

That library is built from exploration traces rather than from the evaluation tasks, but the two are not formally separated. A second configuration separates them explicitly, learning from 26 of the 129 tasks and evaluating on the remaining 103: WEBACT solves 29/103 against 21/103, a 38.1% relative gain, using 6.36 calls per task against 8.90 (Appendix E).

Learning during evaluation completes more of the stream. A third configuration keeps building operations while it works through the 129 tasks, instead of freezing the library in advance. It starts with the 34 operations from the frozen setting, adds 37 more as tasks are solved, and completes 57/129 against the baseline's 41/129, with 66 of its 102 invocations accepted. The two cumulative success curves separate within the first 20 tasks and never meet again (Figure 4a). Both systems run the tasks in the same order, so we can compare them task by task to see where the token savings come from (Figure 4b). Mean tokens fall from 963.4K to 574.7K on the 68 tasks both systems fail

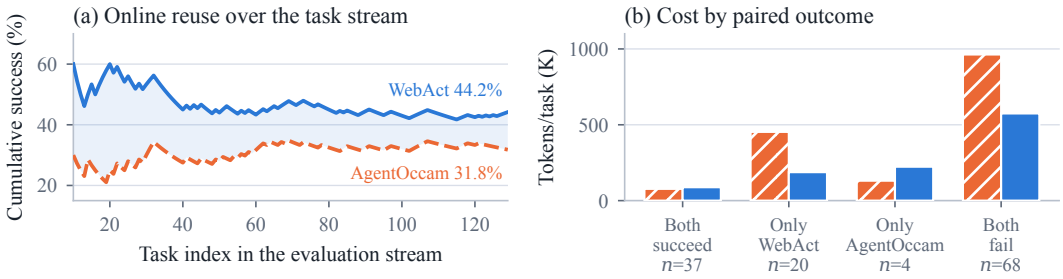

Figure 4: Continued acquisition on the 129-task live stream. (a) Cumulative success separates early and stays separated; the first ten tasks are omitted because a cumulative rate over so few tasks moves by $1/n$ per outcome. (b) Mean tokens per task within each paired-outcome group. The aggregate saving comes from tasks both systems fail and from tasks WEBACT newly solves, not from making shared successes cheaper.

and from 453.2K to 187.6K on the 20 that only WEBACT solves, while rising from 77.6K to 88.3K on the 37 that both solve. The savings therefore come from spending less on tasks that go badly, not from making successful tasks cheaper. We report this configuration as a whole and do not separate what the 37 new operations contribute from what the original 34 contribute.

If we divide the stream into seven consecutive blocks of roughly 20 tasks, WEBACT matches or beats the baseline in every block, and improves on eight of the sites. Success and acceptance rise and fall with the mix of sites in each block; they do not climb steadily as the library grows (Appendix E). This is what we would expect if what matters is whether the library contains a routine for the interaction the current task needs, rather than how many routines it contains.

4.5 RUNTIME CONDITIONS AND THEIR EFFECT ON TASK OUTCOME

Every operation we add is also a new way for the agent to act wrongly, so we test what each runtime condition contributes. On the Reddit evaluation set we disable one control at a time: the invocation condition, the effect check, or the preference updates. A fourth setting weakens target finding, making the agent reuse the element identifiers recorded when the operation was learned (Table 2).

Table 2: Removing one runtime control at a time, on the 85 Reddit evaluation tasks. The last three variants accept no learned call at all, yet differ by 29 tasks in what they complete. Acceptance is not comparable for the second row, since removing the effect check is what the acceptance criterion tests.

Variant	Task success	Executor acceptance
Full WEBACT	**37/85 (43.53%)**	14/16
Without the effect check	24/85 (28.24%)	49/50
Without the invocation condition	0/85 (0.00%)	0/193
With weakened target finding	28/85 (32.94%)	0/61
Without preference updates	29/85 (34.12%)	0/57

Both learned conditions matter for task completion. Removing the invocation condition reduces success from 37/85 to 0/85. Removing the effect check reduces it to 24/85, even though almost all of its calls are accepted, 49 of 50, because without that check the executor has less to verify before accepting.

Acceptance and task completion measure different things. The last three settings in Table 2 accept no learned call at all, and yet they complete 0, 28, and 29 tasks. What distinguishes them is the stage at which their calls fail. Without the invocation condition, all 193 calls record a condition that was not satisfied, a learned action is the planner's first move in 68 of 85 tasks, and the task summaries show forum-page requests invoking product-search routines that fail and lead the agent to stop early. With weakened target finding the calls fail one stage later, 59 of 61 because no target could be found, and 28 tasks still succeed, 15 of them among the 40 that contain a rejected call. With frozen preferences all 57 calls are rejected, 56 at the effect check, and 29 tasks succeed, 14 of them

among the 42 with a rejected call. A rejected call therefore does not mean a lost task, and which condition is missing matters more than how many calls are accepted.

A plan and its check do not always generalize together. In the frozen-preference setting, one search operation accounts for 21 rejections across 16 tasks. Each call reaches a URL matching the query it was given, so the plan handled the new argument correctly, but the effect check still requires the exact query recorded when the operation was learned, so every call is rejected. Both fields come from the same trace segment while describing different things: the plan says what to do with an argument, the check describes a page the agent saw once. Making the plan work for new arguments does not make the check work for them.

Each part of a contract therefore fails in its own way: the invocation condition decides whether a routine is used at all, target finding whether it can act on the page in front of it, and the effect check whether its result is accepted. Separating the two learning signals, operations learned from successes alone and constraints learned from failures alone each solve 35/85 tasks, operations alone use 20.9% fewer calls, and the two together reach 37/85 (Table 13).

5 Related Work

Learning from agent experience. Reflection, revision, and search use experience to improve later decisions without changing what the agent can execute (Shinn et al., 2023; Madaan et al., 2023; Zhou et al., 2024a; Yao et al., 2023). Agent Workflow Memory induces reusable workflows and explores exposing them as callable actions, AutoGuide extracts state-conditioned guidance from successful and failed experiences, and AgentOccam simplifies observations and primitives and supplies our baseline (Wang et al., 2025b; Fu et al., 2024; Yang et al., 2025). These methods either improve how the agent reasons over a fixed action set, or store advice the planner reads; in WebAct the conditions of use are executable parts of the action itself.

Tools and executable skills. Toolformer, tool-making agents, Voyager, and CodeAct establish learning and invoking tools as a way to extend agent capabilities (Schick et al., 2023; Cai et al., 2024; Wang et al., 2024a;b). For an agent acting in an environment, the question becomes how a learned skill is validated. ASI verifies induced skills by executing rewritten trajectories and checking task completion, skill usage, and environment changes (Wang et al., 2025a). On the web, WALT combines parameterized scripts, alternative selectors, validation, and fallback, while SkillWeaver learns Python skills with usage logs and pre-condition filtering (Prabhu et al., 2026; Zheng et al., 2025). ContractSkill repairs the failure its checks localize, WebXSkill adds step-level recovery guidance, and SGDR retrieves skills by page state (Lu et al., 2026; Wang et al., 2026; Li et al., 2026).

Verification is therefore not what distinguishes this work from its neighbors. The closest is ContractSkill, whose checks belong to one skill, so a repair stays inside that skill. In WebAct the result of a check is information about the whole interface: a successful trace adds an operation, a failed attempt adds a constraint that blocks an action pattern in the context where it failed, and every accepted or rejected call changes which operations the planner prefers. Success and failure therefore change different parts of one interface, rather than improving each skill on its own (Table 16).

6 Conclusion

WebAct turns interaction experience into an action interface that keeps changing. Successful traces supply new routines, failed attempts supply constraints on future choices, and the outcome of each invocation guides later reuse. Experiments show that both of the learned conditions matter directly for task success. The complete interface, learned in advance and then frozen, completes more tasks at lower cost on real-world web automation tasks, and the gain survives changes in which tasks it was learned from, a stronger backbone, and the move to live websites. Our work highlights that web agents should learn operations together with their conditions of use.

Reproducibility Statement

Sections 2 and 3 specify the action representation and learning procedure. Evaluation protocols appear in Section 4; Appendix B provides implementation details and the additional experimental tables document task counts, warm-up construction, and evaluation settings. Code is available at `https://anonymous.4open.science/r/Webact_code_commit-D838/`.

Ethics Statement

Reusable browser actions can reduce repetitive work, but can also amplify unintended interactions. Applicability and post-condition checks provide execution evidence, not user authorization or a general safety guarantee. Deployment should respect access controls, website policies, and user permissions, particularly for actions that modify external state.

AI Use Statement

We used generative AI tools to assist with manuscript restructuring, language editing, LaTeX formatting, and integration and interpretation of previously collected experimental results. Language models also serve as components of the web-agent methods evaluated in this work, as described in the methodology. The authors are responsible for the final text, experimental claims, and artifacts.

References

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations*, 2024.

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

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. Autoguide: Automated generation and selection of state-aware guidelines for large language model agents. *arXiv preprint arXiv:2403.08978*, 2024.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6864–6890, 2024.

Jiaxi Li, Ke Deng, Yun Wang, Jingyuan Huang, Yucheng Shi, Qiaoyu Tan, Jin Lu, and Ninghao Liu. Online skill learning for web agents via state-grounded dynamic retrieval. *arXiv preprint arXiv:2606.04391*, 2026. URL `https://arxiv.org/abs/2606.04391`.

Zijian Lu, Yiping Zuo, Yupeng Nie, Xin He, Weibei Fan, Lianyong Qi, and Shi Jin. ContractSkill: Repairable contract-based skills for multimodal web agents. *arXiv preprint arXiv:2603.20340*, 2026. URL `https://arxiv.org/abs/2603.20340`.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, 2023.

Viraj Prabhu, Yutong Dai, Matthew Fernandez, Jing Gu, Krithika Ramakrishnan, Yanqi Luo, Silvio Savarese, Caiming Xiong, Junnan Li, Zeyuan Chen, and Ran Xu. Walt: Web agents that learn tools. In *International Conference on Learning Representations*, 2026. URL `https://openreview.net/forum?id=cgIDqcJcoI`.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, 2024b.

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

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. *arXiv preprint arXiv:2504.06821*, 2025a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 63897–63911. PMLR, 2025b.

Ke Yang, Yao Liu, Sapana Chaudhary, Rasool Fakoor, Pratik Chaudhari, George Karypis, and Huzefa Rangwala. Agentoccam: A simple yet strong baseline for llm-based web agents. In *The Thirteenth International Conference on Learning Representations*, 2025.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, 2023.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. GPT-4V(ision) is a generalist web agent, if grounded. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 61349–61385. PMLR, 2024.

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

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160. PMLR, 2024a.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, 2024b.

A Additional Algorithm Details

Algorithm 2 LIFTSUCCESS: induce contracts from a successful trace.

Require: Successful trace τ with its page observations
$\mathcal{A} \leftarrow \emptyset$
for each segment $\tau^{(k)} \in \text{SEGMENT}(\tau)$ **do**
    Extract typed arguments and construct a parameterized primitive plan
    Bind arguments to plan inputs; generate the operation's semantic description
    Derive `Locate`, `Pre`, `Post` from available segment observations
    Assemble the structured operation α
    Check argument definitions, placeholders, types, and plan bindings; repair inconsistencies
    Initialize $s(\alpha) \leftarrow 1$; add α to $\mathcal{A}$
return $\mathcal{A}$

Algorithm 3 converts failed traces into reusable pruning constraints.

Algorithm 3 EXTRACTCONSTRAINTS converts a failed trace into pruning constraints.

Require: Failed trace τ with outcome $y{=}0$
Ensure: Constraints $\{c_j\}$ where $c = \langle \phi(s), \pi(a, \texttt{tgt}) \rangle$
$\mathcal{C} \leftarrow \emptyset$
if ISREADINESSFAILURE(τ) **then**
    $\phi(s) \leftarrow$ Predicate for an unstable observable state recorded in τ
    $\pi(a, \texttt{tgt}) \leftarrow$ Action pattern to prune, expressed over primitives and target descriptor classes in τ
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{\langle \phi, \pi \rangle\}$
if ISDISAMBIGUATIONFAILURE(τ) **then**
    $\phi(s) \leftarrow$ Predicate for a context anchor recorded in τ
    $\pi(a, \texttt{tgt}) \leftarrow$ Pattern that avoids the failing descriptor class under the anchor in τ
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{\langle \phi, \pi \rangle\}$
return $\mathcal{C}$

B Additional Implementation Details

We use a shared execution setup for AgentOccam and WEBACT. For confidence-based ActSpec selection, we set $s_{\max} = 10$ and $\lambda = 2$. After task-level feedback is accumulated, a separate retirement rule disables an operation after at least three failures in fewer than five uses, or when its failure rate reaches 40% after at least five uses. Disabled operations are excluded from subsequent library queries. This rule remains active when confidence updates are ablated. The full WEBACT setting enables both ActSpec reuse and negative constraints; the two ablations disable one of these signals while keeping the same task stream and warm-up setting.

Structured induction. The generator converts each trace segment into a structured operation with a semantic description, typed arguments, and a primitive plan. It replaces task-specific inputs with arguments and records their bindings to plan steps. The same segment supplies entry/exit states and target descriptors for `Pre`, `Post`, and `Locate`. Rule-based validation checks that bindings refer to defined arguments, arguments occur in the plan or its bindings, argument types agree with their uses, and bindings refer to valid plan steps. Structural repair addresses detected inconsistencies. Runtime target resolution grounds the resulting operation on the current page.

Constructing the checks. Applicability predicates combine the source page's route pattern with the elements and regions used by the trace segment. Element descriptions use accessibility roles, labels, and visible text; modal state and readiness provide additional context. Outcome predicates come from comparing the segment's entry and exit observations: route transitions, elements appearing or disappearing, new text, region updates, and modal changes. The language model refines these

Table 3: Frozen-interface evaluation on five WebArena applications (613 held-out tasks). Costs are per evaluation task and exclude warm-up induction. Best results are in **bold**.

App	Method	Task success ↑	Calls/task ↓	Tokens (K) ↓	Executor accept. ↑
GitLab	AgentOccam	38/144 (26.39%)	11.92	973.4	–
	WEBACT	**54/144 (37.50%)**	**9.76**	**306.4**	39/45 (86.67%)
Map	AgentOccam	21/88 (23.86%)	**9.45**	**275.6**	–
	WEBACT	**26/88 (29.55%)**	9.81	592.5	26/26 (100.00%)
Reddit	AgentOccam	34/85 (40.00%)	10.29	665.1	–
	WEBACT	**37/85 (43.53%)**	**8.66**	**648.4**	14/16 (87.50%)
Shopping	AgentOccam	25/150 (16.67%)	9.74	845.1	–
	WEBACT	**42/150 (28.00%)**	**9.11**	**538.4**	34/46 (73.91%)
Shopping-admin	AgentOccam	7/146 (4.79%)	12.68	625.1	–
	WEBACT	**7/146 (4.79%)**	**7.54**	**125.9**	36/42 (85.71%)
Overall	AgentOccam	125/613 (20.39%)	10.99	716.1	–
	WEBACT	**166/613 (27.08%)**	**8.93**	**408.7**	149/175 (85.14%)

observation-derived candidates using the action sequence and page context. These predicates describe observable conditions for the interaction; their strength depends on which effects the observations expose.

Example. For `Search(query)`, the locator tries role/label, nearby text, and header containment. The pre-condition requires an enabled visible input; the plan enters and submits the bound query; the post-condition checks a refreshed result region or a URL/header reflecting the query. These checks supply evidence about the local interaction. The task evaluator checks the final answer.

Constraint representation. Each negative constraint has the form $\langle \phi(s), \pi(a, \texttt{tgt}) \rangle$. In implementation, ϕ is a conjunction over route, modal or region state, and readiness indicators; π is a pattern over primitive action type and target descriptor class. Readiness constraints require additional waiting or state stabilization before repeating an action. Disambiguation constraints suppress a target descriptor under the same context anchor when repeated selections fail to produce the expected state change. The planner applies these structured constraints directly to candidate actions.

C IN-DEPTH EVALUATION ANALYSIS

Table 3 gives the complete per-application breakdown of the main frozen-interface evaluation summarized in Figure 3, including action calls and executor acceptance.

Table 4: LLM backbone sensitivity on Reddit. Best results are in **bold**.

Backbone	Succ. ↑	Calls/task ↓	Tokens (K) ↓	Executor accept. (%) ↑
Gemini-2.5-Flash	37/85 (43.53%)	8.66	648.4	87.50
Gemini-3-Flash	**38/85 (44.71%)**	8.53	**574.8**	92.86
GPT-5.4 mini	**38/85 (44.71%)**	**8.36**	688.9	**93.75**

Backbone sensitivity. Table 4 studies whether WEBACT depends on a particular LLM backbone, using Reddit tasks and 20% warm-up setting. Gemini-3-Flash and GPT-5.4 mini obtain a slightly higher success rate than Gemini-2.5-Flash. The three backbones execute the learned interface with similar task success and different token costs. This comparison measures interface use across models; the paired GitLab experiment in Appendix F separately evaluates gains over a primitive baseline.

Selected failure cases. An inspected set of 23 site–task pairs where AgentOccam fails and WEBACT succeeds includes 10 budget-exhaustion cases, three target/data-not-found cases, and three looping-

recovery cases; the remaining seven span other termination, quota, validation, and primitive-mismatch failures. A separate invocation subset contains seven locator/executor failures and four post-condition failures. These selected cases locate two sources of benefit: avoiding budget exhaustion and escaping repeated recovery.

D WARM-UP CONSTRUCTION AND ROBUSTNESS

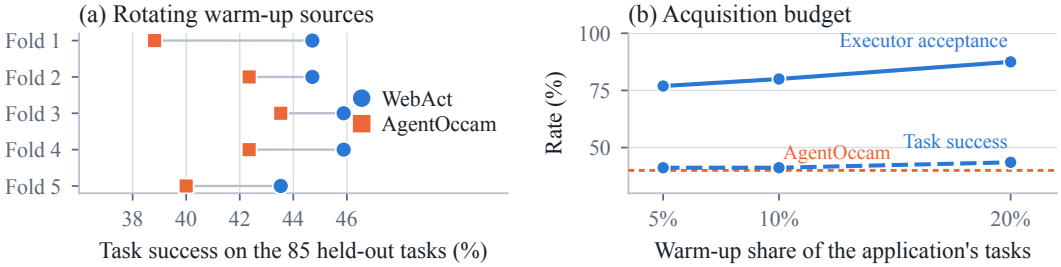

Figure 5: Robustness of the acquired interface on Reddit, summarizing Tables 5 and 6. (a) Five disjoint 21-task warm-up sets each induce a frozen library evaluated on the remaining 85 tasks; WEBACT improves over AgentOccam in every fold. (b) Varying the warm-up share with the evaluation set fixed: task success exceeds the baseline already at 5%, while executor acceptance keeps rising with more source experience.

The Reddit pool contains 106 tasks. Five disjoint sets of 21 tasks rotate as warm-up sources; one task is always held out for evaluation. Each fold therefore evaluates 85 tasks. Fold 5 uses the same 20% warm-up split as the main Reddit experiment. The other four folds partition the remaining eligible tasks. Each fold uses AgentOccam outcomes on the same held-out tasks. Across folds, both the warm-up sources and the overlapping evaluation sets change. WebAct induces a separate frozen library from each warm-up source.

Table 5: Reddit warm-up rotation. Standard deviations describe variation across the five partitions. A is AgentOccam and W is WEBACT; costs are per evaluation task.

Fold	AgentOccam SR	WEBACT SR	A calls / tokens	W calls / tokens
1	33/85 (38.82%)	38/85 (44.71%)	9.44 / 556K	7.80 / 430K
2	36/85 (42.35%)	38/85 (44.71%)	9.47 / 599K	8.78 / 404K
3	37/85 (43.53%)	39/85 (45.88%)	9.51 / 543K	6.81 / 333K
4	36/85 (42.35%)	39/85 (45.88%)	9.28 / 597K	7.69 / 543K
5 (main split)	34/85 (40.00%)	37/85 (43.53%)	10.29 / 665K	8.66 / 648K
Mean $\pm$ std (%)	$41.41\% \pm 1.93\%$	$44.94\% \pm 0.98\%$	9.60 / 592K	7.95 / 472K

Warm-up-size sweep. Table 6 varies the available warm-up tasks while holding the 85-task evaluation set fixed. Larger warm-up sets improve observed executor acceptance and reduce evaluation cost. The 5% setting already exceeds the baseline's 40.00% success, but uses more tokens than AgentOccam (684K versus 665K). The smallest warm-up set therefore improves success before it reduces token cost. The sweep measures execution after acquisition; choosing a deployment budget also requires accounting for acquisition cost.

Table 6: Warm-up sweep on Reddit. Best results are in **bold**.

Method	Warm-up	Succ. ↑	Calls/task ↓	Tokens (K) ↓	Executor accept. ↑
WEBACT	5%	35/85 (41.18%)	9.04	684.0	10/13 (76.92%)
	10%	35/85 (41.18%)	8.81	651.2	16/20 (80.00%)
	20%	**37/85 (43.53%)**	**8.66**	**648.4**	**14/16 (87.50%)**

Template overlap in the main evaluation. Table 7 characterizes the main warm-up and evaluation sets. Task identities are disjoint in every application. Across the 613 evaluation tasks, eight instantiate full task templates also present in warm-up, while 605 (98.7%) use unseen templates. Shopping and Reddit have no shared templates. Reusable interactions can recur across different full task templates; this is the granularity at which an ActSpec can transfer.

Table 7: Full task-template overlap in the main WebArena split. Shared-template tasks instantiate a template present in warm-up.

Application	Warm-up tasks	Evaluation tasks	Shared-template tasks
GitLab	36	144	3
Map	21	88	2
Reddit	21	85	0
Shopping	37	150	0
Shopping-admin	36	146	3
Total	151	613	8

Exact-template separation. The main Reddit partition is template-disjoint: its 21 warm-up tasks and 85 evaluation tasks share no task templates. Table 8 reports an additional frozen evaluation using these same warm-up and evaluation tasks, with the AgentOccam baseline reused. The 38/85 result and the main 37/85 result are evaluations on the same partition, not a comparison of different splits. Different task templates can still share semantic content or interaction routines.

Table 8: Template-disjoint Reddit evaluation.

Method	Task success	Calls/task	Tokens/task	Accepted / invoked
AgentOccam	34/85 (40.00%)	10.29	665,053	–
WEBACT (frozen)	38/85 (44.71%)	7.99	441,217	35/40

E ADDITIONAL LIVE-WEBSITE RESULTS

Table 9 reports consecutive blocks in the 129-task WebVoyager stream. Learning and reuse begin within the first block, with seven ActSpec additions and 18 invocations. Across the stream, the system adds 37 operators, with 66/102 accepted invocations. Task success and executor acceptance rates need not improve from block to block because the site mix and task difficulty change. The final block has only nine tasks, so its rate is particularly sensitive to individual outcomes.

Table 9: Online WebVoyager learning by consecutive task block. W denotes WEBACT and A denotes AgentOccam.

Tasks	W success	A success	W tokens/task	A tokens/task	Added	Accepted / invoked
1–20	11/20	5/20	226,589	570,782	7	13/18
21–40	6/20	6/20	262,676	430,918	3	11/23
41–60	9/20	8/20	332,701	671,597	8	12/17
61–80	10/20	7/20	490,063	883,789	6	13/15
81–100	7/20	6/20	295,989	593,335	4	5/8
101–120	8/20	8/20	476,659	480,540	3	8/12
121–129	6/9	1/9	589,057	594,608	6	4/9

Lightweight state predicates filter actions for applicability. Aggregate token use stays below AgentOccam's, while higher costs in some later blocks coincide with harder pages and anti-bot interruptions. These workload changes affect both task difficulty and opportunities for reuse. Only successful tasks supply new ActSpecs; invocation feedback then tests those routines in later contexts.

Site coverage. Table 10 reports all 13 sites in the evaluation. WebAct improves task success on eight sites, matches AgentOccam on four, and solves one fewer task on Apple. Gains span calculation, search, scholarly browsing, and other site-specific interactions. The largest gain occurs on WolframAlpha, where repeated calculation routines serve many tasks; the complete site breakdown shows how the benefit varies with the workload.

Table 10: Complete per-site task success in the online WebVoyager evaluation.

Site	Tasks	AgentOccam successes	WEBACT successes	Difference
WolframAlpha	34	22	27	+5
HuggingFace	17	2	5	+3
ArXiv	16	5	6	+1
Google Search	16	0	3	+3
ESPN	10	2	4	+2
Cambridge	9	4	5	+1
Google Maps	9	2	3	+1
Apple	7	2	1	−1
Allrecipes	4	0	0	0
BBC News	2	0	0	0
Booking	2	2	2	0
Coursera	2	0	1	+1
Amazon	1	0	0	0
Total	129	41	57	+16

Held-out live-web evaluation. We additionally partition the 129-task set into 26 warm-up tasks and 103 evaluation tasks, with no overlap. WebAct acquires its interface only from warm-up and freezes induction for evaluation, using Gemini-2.5-Flash and the same 15-step budget. AgentOccam outcomes are taken from the main baseline on exactly these 103 tasks. WebAct solves 29/103 tasks versus 21/103, with 655 versus 917 primitive calls and mean token use of 483,038 versus 712,423. Executor acceptance is 35/38. This experiment tests reuse on held-out live tasks; the online experiment evaluates continued acquisition across the full task stream.

Paired task outcomes and token use. Table 11 joins the online WebAct and AgentOccam results on the same 129 tasks. The groups reproduce 57 and 41 total successes, respectively. Both-failure tasks account for approximately 85.3% of the net token saving, and newly solved tasks contribute another 17.1%; increased spending in the other groups offsets part of these savings. These are descriptive groups defined by the two observed outcomes. They locate the aggregate cost difference without attributing it to a single component.

Table 11: Paired WebVoyager outcomes and mean tokens per task (K).

Outcome group	Tasks	AgentOccam	WEBACT
Both succeed	37	77.6	88.3
Only WEBACT succeeds	20	453.2	187.6
Only AgentOccam succeeds	4	131.4	223.5
Both fail	68	963.4	574.7
All tasks	129	604.4	364.3

Frozen-library usage. The frozen WebVoyager library contains six ActSpecs induced from SkillWeaver exploration traces and remains unchanged throughout the 129 evaluation tasks. Of 88 accepted invocations, 40 use a WolframAlpha calculation routine and 39 use a Google search routine. Thus much of the observed reuse comes from two recurring interactions. The library improves aggregate task success without covering every site function, illustrating how a small set of frequently useful routines can contribute to a broader task stream.

Baseline failure categories. Table 12 summarizes selected failure categories for WALT and SkillWeaver. Premature completion and incorrect final answers are common in both summaries, while

budget exhaustion and timeouts identify additional execution difficulties. The selected categories illustrate distinct obstacles between reaching a page and completing a task.

Table 12: Selected reported failure categories on WebVoyager (not exhaustive).

Category	WALT	SkillWeaver
Premature completion	28	24
Answer does not match the checker	37	31
Step budget exhausted	10	27
Page-reading / runtime timeout	0	14

F Additional Backbone Results

The stronger-backbone experiment uses Gemini-3.5-Flash-Lite, a source/evaluation split of 36/144 GitLab tasks (seed 20260724), and a frozen library of 23 GitLab ActSpecs assembled for this setting. Both methods evaluate the same 144 tasks, giving 56/144 (38.89%) versus 59/144 (40.97%), at 9.15 versus 7.89 calls and 512.4K versus 420.1K tokens per task. Because this configuration differs from the main experiment in its split and its library as well as its backbone, it supports the comparison between the two methods under this setting but does not decompose the distance to AgentOccam's published GitLab rate into per-factor contributions. Attributing that distance would require holding the split and library fixed and varying only the model.

G Additional Ablations and Failure Diagnostics

Table 13: Ablation study on Reddit. Best results are in **bold**.

Variant	**Succ. ↑**	**Calls/task ↓**	**Tokens (K) ↓**
WebAct	**37/85 (43.53%)**	8.66	648.4
WebAct-A (w/o constraints)	35/85 (41.18%)	**8.18**	**605.3**
WebAct-C (w/o ActSpecs)	35/85 (41.18%)	10.34	655.7

The learning-signal ablation keeps the Reddit task stream and 20% warm-up setting fixed. ActSpec-only learning lowers cost but solves two fewer tasks than the full system; constraints alone also solve two fewer tasks. The two-task increment is consistent with complementary roles for reusable execution and failure-based pruning.

Table 14: The three ablations in which no ActSpec call was accepted: removing the invocation condition, weakening target finding, and freezing the preference updates. All evaluate the same 85 Reddit tasks; denominators distinguish tasks from invocations.

Metric	No invocation cond.	Weak target finding	No preference updates
Task successes	0/85	28/85	29/85
Accepted / invoked ActSpecs	0/193	0/61	0/57
Invocations satisfying pre-conditions	0/193	61/61	55/57
Tasks choosing an ActSpec first	68/85	14/85	19/85
Mean non-ActSpec actions/task	3.20	8.68	8.42
Target-not-found faults	166	59	1
Failed post-condition faults	26	2	56
Trajectories containing a blank page	16	0	0
Tasks containing rejected calls	70	40	42
Successful tasks among those	0/70	15/40	14/42

Reading the diagnostics. Table 14 reports the invocation-level diagnostics for the three variants with no accepted calls. Without the invocation condition, the 193 failures comprise 166 target-not-found faults, 26 post-condition failures, and one non-interactable target. With weakened target finding,

the invocation condition, effect check, and preference updates all remain enabled; only target finding changes, reusing identifiers recorded at induction rather than resolving targets on the current page. All 61 invocations satisfy the invocation condition, but 59 fail to find their targets. Fifteen of the 40 tasks with rejected invocations ultimately succeed. Without preference updates, 55/57 invocations satisfy the invocation-condition diagnostic; the remaining two occur within a single task. Applicability diagnostics and failure-stage counts describe different properties of the same invocations; they are not disjoint categories. The page can change between candidate filtering and execution-time checks.

Task-level outcomes after rejected calls. In the no-confidence-update variant, 42 of 85 tasks contain at least one rejected ActSpec invocation. Fourteen of those 42 tasks ultimately succeed; the remaining 15 task successes occur among the 43 tasks with no ActSpec invocation. These counts reproduce the aggregate 29/85 task successes and 0/57 accepted calls. For task 621, the first search invocation is rejected by its URL post-condition; the agent then uses clicks and text entry to navigate to the requested forum and create the requested post. Task 733 instead terminates successfully immediately after a rejected invocation because the requested text is already present. The overlap thus establishes that rejection and final success can coexist within a task, while the trajectories distinguish subsequent primitive execution from immediate completion. It does not establish that every rejected invocation failed to produce a useful effect.

Repeated selection of restrictive contracts. The 57 no-confidence invocations use 23 distinct operations. One search operation accounts for 21 calls across 16 tasks, all with satisfied applicability diagnostics. Each reaches a search URL whose query matches the supplied argument, including "earthporn", "DIY tools", and "technology". Its outcome predicate nevertheless requires the fixed query "Iphone 14", rejecting all 21 calls. In task 729, the operation is invoked four times with "UpliftingNews" and receives the same rejection each time. This is a parameter-generalization failure of the check: the input changes in the executed search but remains fixed in the expected outcome. Repeated rejection supplies information for changing selection preferences. These records explain how contract choice affects aggregate acceptance; they do not isolate the effect of confidence updates from other changes in the selected trajectories.

A recorded contract and its verdict. The stored contract for task 621 requires the home-page context and a search box labeled "Search query". Its plan binds the input argument to a text-entry step followed by clicks. The invoked argument is "relations". The recorded outcome predicate expects the search-results route for that query, whereas execution reaches a particular forum post. This mismatch explains the URL rejection directly: the resulting page is beyond the search-results state specified by the contract. Other stored outcome predicates include source-specific result headings and text. The example exposes a second generalization requirement alongside reusable execution: the learned checks must describe the reusable interaction outcome rather than incidental details of the source page.

All 52 source operators carry Reddit site context. Product-search and event-creation refer to their induced descriptions in the examples below. Confidence scores rank actions; a separate usage/failure rule can retire them even in the no-confidence variant. The recorded active library shrinks from 52 to 45 in no-pre and from 52 to 50 in no-confidence. Retirement therefore continues to change available actions even when confidence updates stop.

Illustrative task summaries. The following tasks illustrate how action choices affect subsequent planning:

- **Task 595 (forum navigation).** Product-search and product-selection routines fail with an empty target match; the agent subsequently stops without an answer.
- **Task 598.** Repeated selection of incompatible search/product routines fails to locate targets; the agent stops instead of continuing primitive exploration.
- **Task 580 (forum creation).** An event-creation routine fails its post-condition and the agent stops. This task succeeds in the no-confidence-update run, illustrating the different failure patterns on the same task.

- **Task 600.** The agent interprets a menu of search/navigation ActSpecs as lacking a posting capability, illustrating how the learned menu can shape perceived task feasibility even though primitives remain available.

These examples support the hypothesis that unsuitable learned actions can bias sub-goal selection and stopping. The shared pattern is that execution failure leaves the planner following an unhelpful interpretation of the task.

H Additional Evaluation Settings

Task and invocation outcomes. Task success measures whether the final outcome satisfies the task evaluator. Executor acceptance records the executor's success verdict for an invocation. A task can succeed after a failed invocation through subsequent primitive actions, while a successful invocation may complete only one part of the task.

Table 15: WebAct evaluation configurations. All use the default Gemini-2.5-Flash backbone and 15-decision attempt budget. Frozen evaluation stops induction; confidence updates and retirement remain active.

Evaluation	Interface at evaluation start	Tasks	Induction
WebArena main	Learned from 151 warm-up tasks across five applications	613	Frozen
WebVoyager frozen	Six ActSpecs from SkillWeaver exploration traces	129	Frozen
WebVoyager held-out	Learned from 26 warm-up tasks	103	Frozen
WebVoyager online	34 initial ActSpecs	129	Continues; 37 added

Library provenance and baseline configuration. Table 15 summarizes the four evaluation configurations. WebAct and SkillWeaver use frozen libraries learned from the same offline exploration traces. WebAct induces six ActSpecs and adds none during evaluation; SkillWeaver uses its learned skill library. Both evaluate the same 129 tasks, obtaining 46/129 and 26/129 successes, respectively. Sharing the source traces equalizes the exploration available to each method; the systems still differ in induction, retrieval, and execution, so the comparison bounds the role of exploration volume rather than isolating any single design element. Online evaluation begins with the same 34 ActSpecs that the frozen live-web configuration carries into evaluation, adds 37 more as tasks are solved, and retires operations according to invocation feedback. We report the outcome of this configuration as a whole; the experiment does not separate what the 37 added operations contribute from what the initial 34 contribute.

Execution budget and feedback. The execution budget treats an ActSpec invocation as one decision step. Reported action counters exclude its internal browser operations. For WebVoyager, the counters also exclude the ActSpec decision itself: Calls/task measures primitive decisions, with learned-action invocations reported separately. Tokens measure the reported model usage during evaluation. WebAct processes confidence updates after each task and retains them in the library. Zero confidence lowers an action's rank. Accumulated usage and failure statistics govern retirement. Disabling score updates leaves that separate retirement mechanism enabled.

I Representations and Verification in Related Skill Systems

Table 16: Mechanism-level comparison of learned representations and their verification.

System	Learned representation	Checking and use of feedback
ASI (Wang et al., 2025a)	Programmatic skills induced from successful trajectories.	Validates induced skills through rewritten trajectories, checking task completion, skill use, and environment changes.
SkillWeaver (Zheng et al., 2025)	Python APIs synthesized and honed through exploration.	Tests and improves APIs using environment feedback; filters APIs by relevance and pre-conditions during task execution.
WALT (Prabhu et al., 2026)	Parameterized tool scripts with input schemas and descriptions of pre-conditions and expected outcomes.	Validates tools end-to-end on test inputs and refines them from execution feedback; an external verifier assesses task outcomes.
ContractSkill (Lu et al., 2026)	Executable skill artifacts with pre/post conditions, step structure, and recovery rules.	Deterministic checks localize execution faults; minimal patches repair the implicated component for subsequent execution and transfer.
WebAct	Parameterized primitive plans with explicit target-resolution rules and observation-derived applicability and effect predicates.	Evaluates current-page applicability and local outcome predicates during invocation; recorded outcomes update preference and retirement, while failed attempts supply constraints.

The ActSpec contract places the plan and the observations used to judge its invocation in one structured representation. These are jointly constructed from a trace segment and can be inspected separately: a reusable plan may still carry an overly specific outcome predicate, as the recorded example in Appendix G demonstrates. The component experiments examine the role of these controls within WebAct; the comparison above locates that design relative to existing skill-learning mechanisms.

J Extension to Visual Observations

The contract separates an action's interface from the observation modality; the experiments here use text/accessibility observations. A visual implementation could resolve targets with screenshot anchors and a GUI grounding model, check effects with OCR or visual comparisons, and execute coordinate-level actions. Negative constraints could pair visual context with actions to avoid. Evaluating this extension would require measuring grounding and verification errors and their computational costs.