TRAJSMITH: REUSABLE WEB TOOLS FROM TRAJECTORIES THROUGH EXPLORATION AND VERIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Web agents repeatedly plan browser procedures that earlier trajectories already contain. This repetition increases inference cost and exposes smaller models to errors over long action sequences. Converting trajectories into reusable tools can reduce these demands, but a working source trajectory does not establish when the resulting tool will apply on a changing website. We present **TrajSmith**, a trajectory-based method for synthesizing web tools with explicit reuse conditions. A strong model actively explores the structure around a successful or failed trajectory, inspects related controls and input options, and compiles the observations into atomic and composite tools. An applicability stage challenges candidate tools with changed arguments, boundary cases, neighboring intents, and empty-result queries. Structural checks and an expected result type screen the returned evidence, and a Tool-Use Contract records argument domains, use conditions, and rejection rules. At deployment, a weak model reads the contract to select a tool and fill its arguments; contract checks determine whether the proposal proceeds to execution or fallback. On a 159-task Online-Mind2Web subset selected for tool coverage, the better of two paired runs raises Qwen3-4B success from 18.2% to 30.8% while using fewer recorded agent-loop steps. On WebArena's 763 single-domain tasks, success increases from 19.8% to 24.5%. Additional shared-budget comparisons show gains on source-overlapping tasks for GPT-4o-mini and Qwen3-VL-4B. These results support reuse of recorded procedures across repeated runs and several weak-model configurations.

1 INTRODUCTION

LLM-based web agents browse shopping, knowledge, and enterprise websites (Yao et al., 2023; Zhou et al., 2024; Deng et al., 2023; He et al., 2024; Lu et al., 2024) through an observe–reason–act loop over Document Object Model (DOM) trees or screenshots. On recurring tasks, this loop repeats work: filtering a product listing is planned from scratch even when a previous run already contains the required steps. Repeated planning consumes computation and introduces errors, particularly for smaller models on long tasks. A stronger model improves individual decisions, but an ordinary agent loop still has to make them again.

Prior work reuses experience in several forms. Reflections and trajectory summaries guide subsequent planning (Shinn et al., 2023; Putta et al., 2024), while replay systems preserve interaction sequences (Cheng et al., 2025; Wang et al., 2026a). Executable skill libraries, including Voyager (Wang et al., 2023), take reuse beyond planner context. Web tool learners such as WALT, SkillWeaver, and ContractSkill synthesize callable procedures and use execution feedback or verification to refine them (Prabhu et al., 2026; Zheng et al., 2025; Lu et al., 2026). A remaining question is how to connect the range advertised by a parameterized tool to evidence from the live page. Successful execution on a source task establishes one working instance; reuse also requires deciding which other arguments and neighboring intents the tool can handle.

For a weak model, a callable procedure alone does not resolve this decision. The model must determine whether a retrieved tool fits the task and whether its output is useful. Strong-to-weak reuse therefore benefits from an interface that encapsulates execution details and states both applicability

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.

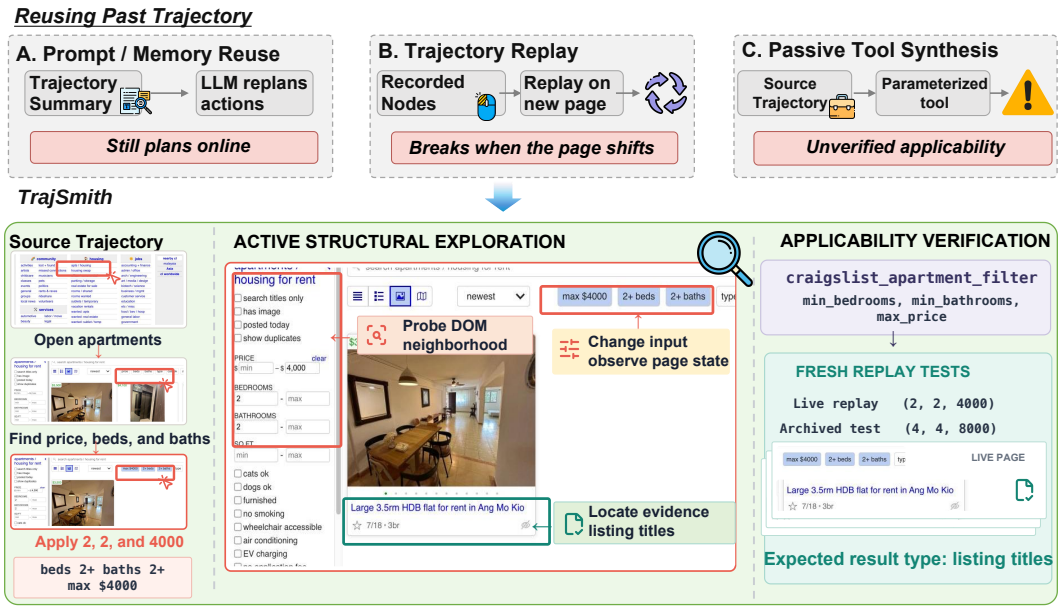


Figure 1: *TrajSmith separates offline tool construction from online reuse.* A strong model explores the page around a recorded trajectory, synthesizes reusable procedures, and screens their applicability. Each tool carries a Tool-Use Contract with argument domains, use conditions, and an expected result type. A weak model reads the contract to select a tool and fill its arguments. The diagram illustrates the progression from stored experience to executable tools with explicit reuse conditions.

and an *expected result type*, such as a numeric statistic or a list of flights. The weak model can then focus on two local decisions: recognizing a suitable tool and extracting its arguments.

To construct this interface, TrajSmith synthesizes tools by active structural exploration. A trajectory records one path through a page with many available actions. Compiling only the recorded nodes ties reuse to that instance. Anchored to the trajectory, a strong model instead probes the DOM neighborhood around each step, including repeated containers and related controls. It uses this structure to construct semantic locators and identify candidate argument values, then turns the trajectory into a family of atomic and composite tools. Correct segments can be retained even when the complete source task fails. Each extension introduces behavior that the source trajectory alone cannot validate.

An applicability challenge stage tests these extensions and records the outcome in a *Tool-Use Contract*. The challenger varies arguments, probes neighboring out-of-scope intents and boundary values, and tests empty-result queries on the live site. Structural checks and a model-authored expected result type screen the returned evidence; changes across arguments help detect tools that repeatedly extract unrelated page content. These applicability checks do not require a reference answer for each generated variant. The resulting contract exposes observed argument values and rejection rules to the online agent. At deployment, the weak model proposes a tool and its arguments, and contract checks screen the proposal before execution or fallback.

We evaluate on the live web with Online-Mind2Web (Xue et al., 2025) and in the reproducible WebArena environment (Zhou et al., 2024). On 159 Online-Mind2Web tasks, the better of two paired runs improves Qwen3-4B success from 18.2% to 30.8%, while reducing recorded agent-loop steps. On WebArena's 763 single-domain tasks, success rises from 19.8% to 24.5%. Most live-web evaluation tasks also supplied trajectories for tool construction, so these runs measure reuse on previously attempted tasks. Further comparisons impose shared call and action caps and test reuse with additional weak models. Our contributions are:

- A trajectory-anchored synthesis procedure that probes page structure and compiles reusable atomic and composite tools from complete trajectories and correct sub-trajectories.

- An applicability-testing procedure that challenges candidate tools on the live site and records argument domains, rejection rules, and expected result types in a Tool-Use Contract.
- An evaluation of contract-guided strong-to-weak reuse on live and self-hosted websites, with repeat runs, shared-budget comparisons, component analyses, and worked deployment cases.

2 Related Work

Web and GUI agents. LLM-based web and GUI agents follow an observe–plan–act loop over a page (Yao et al., 2023; Gur et al., 2023; He et al., 2024; Lai et al., 2024; Cheng et al., 2024; Hong et al., 2024), evaluated on realistic-navigation benchmarks (Zhou et al., 2024; Deng et al., 2023; Yao et al., 2022; Lu et al., 2024; Koh et al., 2024) and, more recently, open-web and long-horizon or generated environments (Gupta et al., 2026; Jang et al., 2026; Yuan et al., 2026; Zhou, 2026; Hao et al., 2026). These systems provide the observation and action interfaces for general web tasks. TrajSmith builds on those interfaces to reuse recurring procedures through executable tools.

Tool use and self-improving web procedures. Beyond invoking external tools (Schick et al., 2023; Cai et al., 2023), web and GUI agents increasingly induce reusable procedures from their own interaction, synthesizing callable site tools, programmatic skills, workflows, or repairable and self-improving skill libraries (Prabhu et al., 2026; Jiang et al., 2026; Wang et al., 2025a;b; Lu et al., 2026; Zheng et al., 2025; Song et al., 2025; Li et al., 2024; Wang et al., 2024; Lou et al., 2026; Chen et al., 2026b; Wang et al., 2026b; Zhao et al., 2026; Zhong et al., 2026). WALT and SkillWeaver learn executable browser skills, and ContractSkill explicitly verifies and repairs skills against contracts. Our focus is the connection between trajectory-anchored exploration, tests of argument applicability, and the interface presented to a weak downstream model. Retrieval uses descriptions to rank candidates, while explicit contract checks screen the selected call.

Other forms of experience reuse. Experience can also guide subsequent decisions through reflections, summaries, and memory (Shinn et al., 2023; Putta et al., 2024; Tan et al., 2026; Yan et al., 2026; Xiao et al., 2026; Shi et al., 2026). Replay, hierarchical memory, and pruning reduce repeated interaction or context costs (Cheng et al., 2025; Nekoei et al., 2025; Wang et al., 2026a; Lin et al., 2026; Zhang et al., 2025). Voyager additionally stores executable code skills (Wang et al., 2023). Distillation methods transfer agent behavior into skills or modules for another model (Ni et al., 2026; Qiu et al., 2025). These approaches motivate reuse at different levels. TrajSmith concentrates on a web-specific callable whose procedure, arguments, and reuse conditions can be exposed separately to the downstream agent.

Verification and reliable reuse. Reusable tools need finer correctness criteria than end-to-end success, and recent work verifies computer-use agents through action-effect checks, intent-conditioned evaluation, verifiers, and safety guardrails (Zhang et al., 2026b; Chen et al., 2026a; Rosset et al., 2026; Zhai et al., 2026; Aghzal et al., 2026; Zhang et al., 2026a; Cui et al., 2026; Chen et al., 2026c; Ding et al., 2026). Skill-induction pipelines isolate verification, gate admission, and repair a skill against a postcondition (Lu et al., 2026). Our applicability tests emphasize changed arguments, boundary cases, and neighboring intents, alongside source-instance replay. This resembles property-based testing and REST-API fuzzing (Atlidakis et al., 2018; Liu et al., 2024), but uses observed page structure and a model-authored output description to construct the tests. The resulting contract records empirically tested scope; its reliability depends on the tests and their output checks. For final live-web task evaluation, we use WebJudge (Xue et al., 2025); its task-level verdict is separate from the applicability check on a generated variant.

3 Method

TrajSmith converts a trajectory into reusable tools $t = \langle p, s \rangle$, pairing an implementation p with a *Tool-Use Contract* s. The method has three stages: offline synthesis, offline applicability verification, and online reuse (Figure 2).

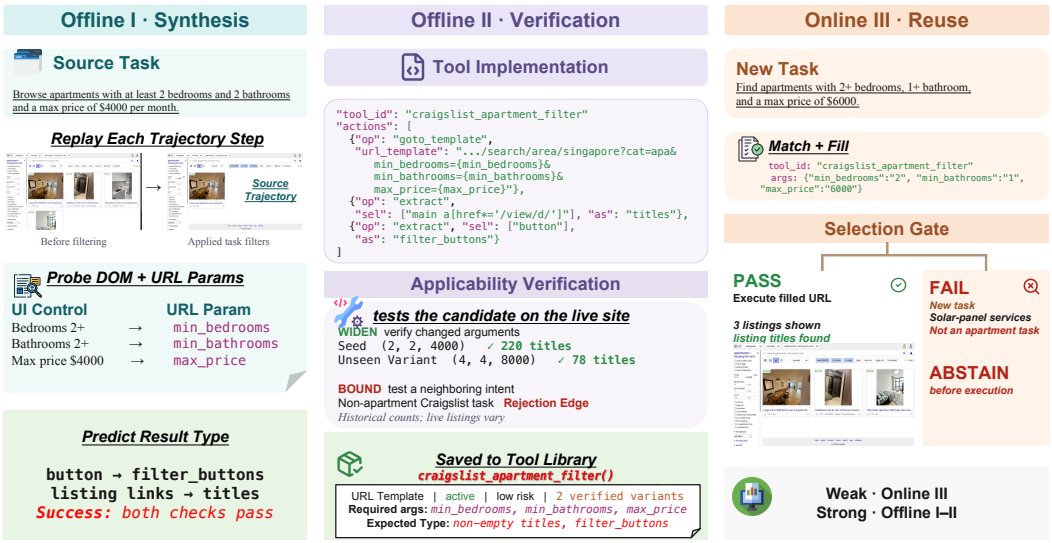


Figure 2: *Overview of TrajSmith.* Offline, a strong model explores the structure surrounding a trajectory, compiles atomic and composite tools, and challenges their applicability on the live site. Online, a weak model selects a tool and fills its arguments from the contract; lifecycle, host, risk, intent, and argument checks determine whether to execute it or fall back. The housing example illustrates how a URL-based procedure, parameter domains, and reuse conditions are represented across these stages.

3.1 Tasks, Tools, and Tool-Use Contracts

Web-agent trajectories. A web agent receives an instruction, observes DOM trees or screenshots, and emits browser actions. TrajSmith accepts recorded trajectories from both successful and failed rollouts. Reuse is defined for individual procedures: a failed task can still contain useful navigation, search, or extraction steps that can be verified independently.

Tool and Tool-Use Contract. We represent a site-specific callable as

$$t = \langle p, s \rangle, \tag{1}$$

where p is its implementation and s specifies how it may be reused. The contract has three groups of fields. *Matching* fields provide use conditions, an argument schema with a value *domain* for each slot, a retrieval description, and a predicted expected result type. The expected type describes the form and content of a valid output. *Governance* fields determine eligibility: lifecycle is `candidate` until admission and `active` afterwards, while risk and generalization tier guide use. *Composition* fields specify a tool's role and the capability or page state it *provides* and *consumes*. Appendix A.4 lists the full schema. Stored argument values carry provenance from page observations and applicability checks. A value is marked observed only after a fresh replay passes structural and expected-type checks. Runtime checks use membership for enumerated domains and may interpolate between numeric endpoints; an interpolated value is admissible but is not claimed to have been individually tested. When several slots are supplied, the gate checks each slot independently and does not assert joint validity unless that combination was exercised during verification. The bound site is derived from navigation for host-based retrieval.

3.2 Tool Synthesis by Active Structural Exploration

Motivation. A recorded path contains only one instance of a page's available actions. Active exploration identifies related controls and parameter choices that support reuse beyond that instance.

Exploration and probing. A strong model replays recorded steps on the live site and probes the structure around acted-on controls. Successfully replayed prefixes preserve intermediate pages and interaction state for local synthesis when later steps fail. Using the DOM and accessibility tree,

exploration inspects enclosing containers, repeated sibling elements, and related pagination or sorting controls. This supports a *family selector*: locators based on roles and stable attributes within repeated structure. Probing also inspects input options, observed URL query parameters, and read-only GET endpoints visible in network traffic. These observations expand the candidate capabilities and parameter domains tested by subsequent verification.

Choosing a stable implementation. Each step uses the simplest execution form supported by the observations: a URL template or read-only backend GET when it reaches the required page or data, semantic UI actions otherwise, and an agent subgoal when direct operations are insufficient. UI steps retain ordered locator fallbacks so that a changed primary locator need not force re-synthesis. Parameterization uses observed task-relevant slots, rather than arbitrary click targets or timing decisions. Synthesis first constructs a fixed seed using the source values to check the action pattern and evidence selector. It then exposes observed input slots as parameters. If proposed evidence selectors miss the terminal page, the model receives the observation and the failed locators and can select again; a candidate without usable evidence is not retained as a verified tool. This ordering separates an execution failure on the original instance from a failure introduced by parameterization. The complete procedure is detailed in Appendix A.1.

A trajectory becomes a family of tools. A trajectory usually contains several reusable actions, which we compile at two levels. Around each control the trajectory acted on, exploration emits an *atomic* tool for the action it *provides* (search, filter, sort, paginate, extract), parameterized over observed values; a *composite* tool then chains these atoms in the trajectory's order to reproduce the end-to-end task. An atom that needs an already-loaded page *consumes* that page and is kept for composition rather than offered standalone. A failed source task can still supply a correct action segment or a fixed replay of the page it reached. Each local tool describes its observed operation and expected output. Fixed query and filter values remain part of its reuse conditions; parameter slots expose the inputs that the procedure can change.

3.3 Applicability Verification: Testing Reuse Conditions

What is verified, and by whom. Synthesis proposes a reusable capability; applicability verification tests that proposal against the live site. Here, "verified" denotes a reproducible operational screen, not a proof of task-level semantic correctness. A challenger reads the public contract and proposes calls that probe the advertised argument range and neighboring intents. This tests reuse conditions beyond successful reproduction of the source trajectory.

The loop. The challenger generates four classes of tests: *in-scope unseen* values that should produce valid results; *out-of-scope* intents that should be rejected; *boundary* values near the claimed range; and *empty-result* queries that should return an empty result rather than unrelated page content. Structural verification uses repeated fresh executions, normally a majority of $N{=}3$ trials. A semantic check separately compares terminal evidence with the expected result type predicted at synthesis. Comparing outputs across arguments also detects query-invariant extraction. These tests screen applicability without reference answers for the generated variants. Passing tests update the stored domain and rejection rules; a tool can remain `candidate`, be admitted with a narrower scope, or retain only fixed-instance reuse. The contract therefore records evidence for operational reuse rather than a guarantee that every accepted argument tuple satisfies the original task semantics. Parameterized tools receive the fuller challenge path, while parameterless tools use lighter replay checks. Repair is bounded and can revise the implementation or contract before rechecking. Appendix A.3 specifies the branches and their stopping rules.

Admission from correct sub-trajectories. Components are checked against their local capabilities, independently of source-task success. Parameterized local tools use generated challenges when available, otherwise repeated structural replay. Admission records outcomes and updates argument values and rejection rules.

Relation to source-task admission and drift recovery. Source replay checks one recorded instance; the challenge stage probes changed arguments and neighboring intents to refine the advertised scope. Structural locators and alternative execution forms address page variation while the observed controls and endpoints remain available.

3.4 Online Tool Selection and Execution

Contract-based retrieval. Before showing tools to the weak model, retrieval filters the library by lifecycle, host, risk permission, and page prerequisites for consuming atoms. A scorer ranks the remaining tools by description relevance, generalization tier, and argument availability. The shortlist includes parameter domains as filling hints. If no candidate remains, the agent continues with ordinary browser actions. The selected proposal then passes the contract checks below.

Weak-model interface. The weak model sees the matching fields of shortlisted tools rather than their implementations. It selects a tool and fills its arguments as JSON. This separates task matching and argument extraction from the browser procedure encoded in the tool, allowing a smaller model to invoke a procedure synthesized by a stronger one.

Selection gate. A weak-model call at temperature 0 proposes one tool and its arguments. For instruction x, tool t, and proposed arguments a, the gate checks

$$\begin{aligned}\mathrm{Pass}(x,t,a) \iff & L(t) \wedge H(x,t) \wedge R(t) \\ & \wedge J(x,t) \wedge A(t,a),\end{aligned} \tag{2}$$

where L checks active lifecycle, H checks the task host, R checks risk permission, J screens intent against contract rejection rules, and A checks required arguments and their declared domains. Lifecycle, host, risk, and argument checks are programmatic; J uses the selector model when rejection rules are present. If a check rejects the proposal, the weak agent falls back to ordinary browser actions. Selection, intent screening, and model-assisted execution use the weak model in the strong-to-weak configuration.

Execution. Execution fills the argument templates and runs the selected procedure. Result selectors, API payloads, or agent answers supply execution evidence. An expected-result check screens this evidence before the agent continues or answers. Unparseable judgments reject the proposal in the shared-budget experiments; the original runs accept them. Rejected or absent evidence leads to weak-agent fallback. A failed tool attempt can still change the page state and consume part of the execution budget. Tool-level evidence and complete task success are distinct, with the latter measured by the benchmark evaluator. Typed failure records associate retrieval, argument, execution, or evidence errors with the responsible tool.

3.5 Implementation

TrajSmith uses Python, an OpenAI-compatible model client, and Playwright-controlled Chromium with accessibility observations from `aria_snapshot`. For Online-Mind2Web, Claude Opus 4.6 performs source rollouts, exploration, synthesis, and applicability verification. Qwen3-4B-Instruct-2507 (Qwen3-4B below), served locally with vLLM, handles online selection, model-assisted execution, and fallback. The shared-budget comparisons reuse the same Claude Opus 4.6 library with GPT-4o-mini or Qwen3-VL-4B-Instruct. WebArena instead uses GPT-5.4 for construction. Section 4.1 specifies task evaluators and budgets; Appendix A details the implementation.

4 Experiments

We evaluate weak-agent reuse on live and self-hosted websites, then examine shared-budget deployment and the mechanisms behind success and failure.

4.1 Setup

Benchmarks. Online-Mind2Web (Xue et al., 2025) contains 300 real-world browser tasks with changing content, dynamic widgets, and access failures. **WebArena** (Zhou et al., 2024) provides self-hosted websites and deterministic task checkers. We evaluate its 763 single-domain tasks across Shopping, Shopping-Admin/CMS, Reddit, GitLab, and Map.

Task sets and construction sources. The live-web collection identified a reachable pool of 194 tasks. The main live-web comparison uses an archived 159-task subset selected for tool coverage. Both conditions run on these same IDs. The frozen Claude Opus 4.6 library contains 896 tools from 190 source tasks, with 236 active entries. Of the 159 evaluation IDs, 155 supplied construction

Table 1: **Weak-agent reuse on Online-Mind2Web ($N = 159$).** The better of two paired runs uses Qwen3-4B and a frozen Claude Opus 4.6 library. Steps count agent-loop iterations; budgets are specified in Section 4.1.

Method	Solved tasks	Success (%)	Mean steps
No-tool	29	18.2	17.01
TrajSmith (ours)	**49**	**30.8**	**14.48**

Table 2: **WebArena success rates (%).** Both conditions use Qwen3-4B-Instruct-2507 and the native checker. TrajSmith adds a frozen GPT-5.4-synthesized library; Δ denotes percentage points.

	Shopping	CMS	Reddit	GitLab	Map	Overall
Tasks (N)	187	182	106	179	109	763
No-tool	20.9	25.3	13.2	19.6	15.6	19.8
+TrajSmith	**24.1**	**31.3**	**16.0**	**25.1**	**21.1**	**24.5**
Δ (pp)	+3.2	+6.0	+2.8	+5.5	+5.5	+4.7

trajectories, so these runs measure reuse on previously attempted tasks. Shared-budget experiments use additional source-overlapping task sets. Appendix C.2 lists the intersections; the baseline comparison and weak-selector ablations use the same 58-task subset.

Protocol and metrics. We score Online-Mind2Web trajectories with the WebJudge protocol (Xue et al., 2025), using Claude Opus 4.6 for the original runs and GPT-4.1 for the shared-budget runs. Both follow the same key-point, screenshot-selection, and final-verdict pipeline, with a screenshot threshold of 3 on a five-point scale. WebArena uses its native deterministic checker. We report task success, paired success changes, and recorded agent-loop steps. Tool calls can contain several browser operations, so these steps measure the agent loop rather than primitive browser actions.

Conditions. The live-web deployment condition uses Claude Opus 4.6-synthesized tools with Qwen3-4B selection and fallback. Each run pairs it with a Qwen3-4B no-tool run on the same task IDs. For the 159-task comparison and the 32-task GPT-4o-mini comparison, the main tables show the better of two completed paired runs by TrajSmith success, retaining its paired control and all associated metrics; both runs are documented in the appendix. The no-tool loop allows 18 steps; the historical tool path allows one routing step plus up to 18 fallback steps. WebArena uses Qwen3-4B-Instruct-2507 at temperature 0 with a 30-step budget and a frozen library synthesized by GPT-5.4. The shared-budget runs cap each online episode at 18 model-call attempts and 30 browser action units, shared across selection, tool internals, verification, and fallback, with a 300-second soft deadline. They use GPT-4o-mini or Qwen3-VL-4B-Instruct throughout the online path. Appendix C.12 specifies the controls, source overlap, and accounting.

4.2 Main Results: Reusing Tools with a Weak Agent

Live web (Online-Mind2Web). Table 1 reports the selected paired run. Tools increase success from $29/159$ to $49/159$, a gain of 12.6 percentage points, while reducing mean recorded agent-loop steps from 17.01 to 14.48. The paired outcomes include 30 gains and 10 losses. A second run improves from $28/159$ to $47/159$; Appendix C.6 reports both runs and their paired statistics.

4.3 WebArena: Reuse across Five Site Categories

Under WebArena's deterministic checker (Table 2), tools improve success in all five site categories, from 19.8% to 24.5% overall: 36 additional solved tasks and a 4.7-point gain. The largest improvements occur on CMS, GitLab, and Map, where reusable filtering, lookup, and navigation procedures are common. WebArena complements the live-web experiment: site state and scoring are controlled, while the offline synthesizer changes from Claude Opus 4.6 to GPT-5.4. The consistent direction across all five categories supports reuse across several kinds of web workflow.

Table 3: **Shared-budget reuse on 32 Online-Mind2Web tasks.** All conditions share the same task IDs and call/action caps; 30 IDs supplied construction trajectories. GPT-4o-mini shows the better of two paired runs; Qwen3-VL-4B has one run.

Online model	Method	Solved tasks	Success (%)
GPT-4o-mini	No-tool	9	28.1
	TrajSmith (ours)	**16**	**50.0**
Qwen3-VL-4B	No-tool	8	25.0
	TrajSmith (ours)	**12**	**37.5**

Table 4: **Baseline comparison on 58 shared Online-Mind2Web tasks.** All methods use Qwen3-4B online and Claude Opus 4.6 WebJudge. Each row reports one method on the complete shared task set.

Method	Solved tasks	Success (%)
No-tool	5	8.6
ContractSkill (adapted)	6	10.3
TrajSmith (ours)	**18**	**31.0**

4.4 Shared-Budget Reuse across Weak Models

Table 3 compares the two weak models on the same 32 tasks under shared call and action caps. GPT-4o-mini improves from $9/32$ to $16/32$, and Qwen3-VL-4B improves from $8/32$ to $12/32$. These are complete-system outcomes, including tool attempts and fallback. Appendix C.12 specifies the execution protocol and separates accepted tool evidence from fallback outcomes.

4.5 Comparison with Tool-Learning Baselines

Table 4 compares methods on the same 58 Online-Mind2Web tasks with Qwen3-4B as the online model. TrajSmith solves 18 tasks (31.0%), compared with six (10.3%) for adapted ContractSkill (Lu et al., 2026) and five (8.6%) without tools. ContractSkill generates and repairs skills per task; TrajSmith retrieves from a frozen library. We restrict the direct table to ContractSkill because it has complete records on these same IDs; archived WALT and SkillWeaver records use different task sets or incomplete protocols and are not mixed into this denominator. Appendix C.11 specifies the construction models, task selection, and interaction budgets.

4.6 Analysis

Component ablations. Table 5 isolates library views and runtime controls on the same 58 tasks with Qwen3-4B. Removing retrieval reduces successes from 18 to 13; removing model-assisted extraction reduces them to 16. Admitting candidate tools yields 17, while removing semantic checking or failed-origin tools raises the count to 19 and 20. Separately rebuilt full and passive/no-verification libraries each solve 18/58 tasks, with unmatched construction costs (Appendix C.9). Retrieval has the largest observed effect among these controls; the independent contributions of exploration and verification remain unresolved. Appendix A.10 defines each intervention.

Where the gain concentrates. Table 6 breaks down the same run as Table 1. Easy tasks gain 14 successes and medium tasks gain seven; hard-task success is similar across the two conditions (6 versus 5). The current library and selection policy help most on easy and medium tasks; the additional repeat has the same pattern (Appendix C.6).

Coverage of weak-agent failures. Of the 53 no-tool failures in the 58-task set, 29 have associated active tools (63 entries). Successful tool-assisted cases include MTA neighborhood maps, NBA statistics, AccuWeather forecasts, and Cookpad recipes.

Table 5: **Component ablations on 58 shared tasks.** All conditions use Qwen3-4B and the same frozen library, modified only by the named library view or runtime control.

Setting	Solved tasks	Success (%)
Full TrajSmith	18	31.0
Include candidate tools	17	29.3
w/o failed-origin tools	**20**	**34.5**
w/o retrieval	13	22.4
w/o semantic verification	19	32.8
w/o model-assisted extraction	16	27.6

Table 6: **Success by task difficulty in the selected 159-task run.** Each entry gives solved tasks and success rate. All rows use the same paired run as Table 1.

Difficulty	Tasks (N)	No-tool	TrajSmith (ours)
Easy	49	9 (18.4%)	**23 (46.9%)**
Medium	71	14 (19.7%)	**21 (29.6%)**
Hard	39	**6 (15.4%)**	5 (12.8%)
Overall	159	29 (18.2%)	**49 (30.8%)**

What the contract contributes. The NBA contract specifies Chris Paul and the current season as its query scope. The Craigslist tool exposes an enumerated listing-condition parameter (Appendix A.9). These contracts specify intent and argument boundaries.

Construction and online cost. The recorded one-time cost of constructing the Claude Opus 4.6 library is $41.25. Online selection and fallback use the weak model. In the selected 159-task run, tools reduce mean agent-loop steps by 14.8%, from 17.01 to 14.48. The shared-budget Qwen3-VL-4B comparison uses 15.63 model calls without tools and 14.22 with tools, including internal calls and retries. Appendix E.1 separates offline construction, online execution, and evaluator costs.

5 Conclusion

TrajSmith turns recorded web trajectories into reusable tools through active structural exploration and applicability testing. Exploration identifies related controls and parameter slots; verification screens the resulting procedures and records their reuse conditions in a Tool-Use Contract. A weak model uses this contract to select and invoke tools while keeping browser execution details inside the callable. Experiments on Online-Mind2Web and WebArena support repeated reuse, with gains across several weak-model configurations. Transfer to other tasks depends on recovering useful page states, validating parameter effects, and selecting procedures that advance the requested goal. Broader parameter coverage and stronger checks of task-specific evidence remain priorities for reuse beyond the construction tasks.

AI Use Statement

AI tools assisted with manuscript editing and translation, literature checks, code development, and experimental design and analysis. The authors take responsibility for the methods, results, and final manuscript.

REFERENCES

Mohamed Aghzal, Gregory J. Stein, and Ziyu Yao. Why do llm-based web agents fail? a hierarchical planning perspective, 2026. URL https://arxiv.org/abs/2603.14248.

Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Rest-ler: automatic intelligent rest api fuzzing. *arXiv preprint arXiv:1806.09739*, 2018.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023.

Rongqian Chen, Yu Li, Zeyu Fang, Sizhe Tang, Weidong Cao, and Tian Lan. Intentscore: Intent-conditioned action evaluation for computer-use agents, 2026a. URL https://arxiv.org/abs/2604.05157.

Tianyi Chen, Yinheng Li, Michael Solodko, Sen Wang, Nan Jiang, Tingyuan Cui, Junheng Hao, Jongwoo Ko, Sara Abdali, Leon Xu, Suzhen Zheng, Hao Fan, Pashmina Cameron, Justin Wagle, and Kazuhito Koishida. Cua-skill: Develop skills for computer using agent, 2026b. URL https://arxiv.org/abs/2601.21123.

Yurun Chen, Zeyi Liao, Ping Yin, Taotao Xie, Keting Yin, and Shengyu Zhang. Safepred: A predictive guardrail for computer-using agents via world models, 2026c. URL https://arxiv.org/abs/2602.01725.

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

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. In *ACL 2024*, 2024. URL https://aclanthology.org/2024.acl-long.505/.

Chaoqun Cui, Jing Huang, Shijing Wang, Liming Zheng, Qingchao Kong, and Zhixiong Zeng. Agentic reward modeling: Verifying gui agent via online proactive interaction, 2026. URL https://arxiv.org/abs/2602.00575.

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 *NeurIPS 2023 Datasets and Benchmarks Track*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/5950bf290a1570ea401bf98882128160-Abstract-Datasets_and_Benchmarks.html.

Xuwei Ding, Skylar Zhai, Linxin Song, Jiate Li, Taiwei Shi, Nicholas Meade, Siva Reddy, Jian Kang, and Jieyu Zhao. The blind spot of agent safety: How benign user instructions expose critical vulnerabilities in computer-use agents, 2026. URL https://arxiv.org/abs/2604.10577.

Tanmay Gupta, Piper Wolters, Zixian Ma, Peter Sushko, Rock Yuren Pang, Diego Llanes, Yue Yang, Taira Anderson, Boyuan Zheng, Zhongzheng Ren, Harsh Trivedi, Taylor Blanton, Caleb Ouellette, Winson Han, Ali Farhadi, and Ranjay Krishna. Molmoweb: Open visual web agent and open data for the open web, 2026. URL https://arxiv.org/abs/2604.08516.

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

Shibo Hao, Zhining Zhang, Zhiqi Liang, Tianyang Liu, Yuheng Zha, Qiyue Gao, Jixuan Chen, Zilong Wang, Zhoujun Cheng, Haoxiang Zhang, Junli Wang, Hexi Jin, Boyuan Zheng, Kun Zhou, Yu Wang, Feng Yao, Licheng Liu, Yijiang Li, Zhifei Li, Zhengtao Han, Pracha Promthaw, Tommaso Cerruti, Xiaohan Fu, Ziqiao Ma, Jingbo Shang, Lianhui Qin, Julian McAuley, Eric P. Xing, Zhengzhong Liu, Rupesh Kumar Srivastava, and Zhiting Hu. Cocoabench: Evaluating unified digital agents in the wild, 2026. URL https://arxiv.org/abs/2604.11201.

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 *ACL 2024*, 2024. URL https://aclanthology.org/2024.acl-long.371/.

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

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

Linxi Jiang, Rui Xi, Zhijie Liu, Shuo Chen, Zhiqiang Lin, and Suman Nath. Web verbs: Typed abstractions for reliable task composition on the agentic web, 2026. URL https://arxiv.org/abs/2602.17245.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *ACL 2024*, 2024. URL https://aclanthology.org/2024.acl-long.50/.

Hanyu Lai, Xiao Liu, Iat Long Iong, Shuntian Yao, Yuxuan Chen, Pengbo Shen, Hao Yu, Hanchen Zhang, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. Autowebglm: A large language model-based web navigating agent. In *KDD 2024*, 2024. URL https://dl.acm.org/doi/10.1145/3637528.3671620.

Yanda Li, Chi Zhang, Wenjia Jiang, Wanqi Yang, Bin Fu, Pei Cheng, Xin Chen, Ling Chen, and Yunchao Wei. Appagent v2: Advanced agent for flexible mobile interactions, 2024. URL https://arxiv.org/abs/2408.11824.

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, 2026. URL https://arxiv.org/abs/2603.24533.

Jinghua Liu, Yi Yang, Kai Chen, and Miaoqian Lin. Generating api parameter security rules with llm for api misuse detection. *arXiv preprint arXiv:2409.09288*, 2024.

Renze Lou, Baolin Peng, Wenlin Yao, Qianhui Wu, Hao Cheng, Suman Nath, Wenpeng Yin, and Jianfeng Gao. The tool illusion: Rethinking tool use in web agents, 2026. URL https://arxiv.org/abs/2604.03465.

Xing Han Lu, Zdenek Kasner, and Siva Reddy. WebLINX: Real-world website navigation with multi-turn dialogue. *arXiv preprint arXiv:2402.05930*, 2024.

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

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

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills, 2026. URL https://arxiv.org/abs/2603.25158.

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 *ICLR 2026*, 2026. URL https://arxiv.org/abs/2510.01524.

Pranav Putta, Edmund Mills, Naman Garg, Sumeet Motwani, Chelsea Finn, Divyansh Garg, and Rafael Rafailov. Agent Q: Advanced reasoning and learning for autonomous AI agents. *arXiv preprint arXiv:2408.07199*, 2024.

Jiahao Qiu, Xinzhe Juan, Yimin Wang, Ling Yang, Xuan Qi, Tongcheng Zhang, Jiacheng Guo, Yifu Lu, Zixin Yao, Hongru Wang, et al. Agentdistill: Training-free agent distillation with generalizable mcp boxes. *arXiv preprint arXiv:2506.14728*, 2025.

Corby Rosset, Pratyusha Sharma, Andrew Zhao, Miguel Gonzalez-Fernandez, and Ahmed Awadallah. The art of building verifiers for computer use agents, 2026. URL https://arxiv.org/abs/2604.06240.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2023.

Yibo Shi, Jungang Li, Linghao Zhang, Zihao Dongfang, Biao Wu, Sicheng Tao, Yibo Yan, Chenxi Qin, Weiting Liu, Zhixin Lin, Hanqian Li, Yu Huang, Song Dai, Yonghua Hei, Yue Ding, Xiang Li, Shikang Wang, Chengdong Xu, Jingqi Liu, Xueying Ma, Zhiwen Zheng, Xiaofei Zhang, Bincheng Wang, Nichen Yang, Jie Wu, Lihua Tian, Chen Li, and Xuming Hu. Androtmem: From interaction trajectories to anchored memory in long-horizon gui agents, 2026. URL https://arxiv.org/abs/2603.18429.

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*, 2023.

Yueqi Song, Frank F. Xu, Shuyan Zhou, and Graham Neubig. Beyond browsing: API-based web agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 11066–11085, 2025.

Yunteng Tan, Zhi Gao, and Xinxiao Wu. Enhancing web agents with a hierarchical memory tree, 2026. URL https://arxiv.org/abs/2603.07024.

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. *arXiv preprint arXiv:2305.16291*, 2023.

Junjie Wang, Zequn Xie, Dan Yang, Jie Feng, Yue Shen, Duolin Sun, Meixiu Long, Yihan Jiao, Zhehao Tan, Jian Wang, Peng Wei, and Jinjie Gu. Webclipper: Efficient evolution of web agents with graph-based trajectory pruning, 2026a. URL https://arxiv.org/abs/2602.12852.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024.

Xiaoce Wang, Guibin Zhang, Junzhe Li, Jinzhe Tu, Chun Li, and Ming Li. Tooltok: Tool tokenization for efficient and generalizable gui agents, 2026b. URL https://arxiv.org/abs/2602.02548.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. In *COLM 2025*, 2025a. URL https://openreview.net/forum?id=lsAY6fWsog.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *ICML 2025*, 2025b. URL https://proceedings.mlr.press/v267/wang25bx.html.

Han Xiao, Guozhi Wang, Hao Wang, Shilong Liu, Yuxiang Chai, Yue Pan, Yufeng Zhou, Xiaoxin Chen, Yafei Wen, and Hongsheng Li. Ui-mem: Self-evolving experience memory for online reinforcement learning in mobile gui agents, 2026. URL https://arxiv.org/abs/2602.05832.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? assessing the current state of web agents. In *COLM 2025*, 2025. URL https://openreview.net/forum?id=6jZi4HSs6o.

Dawei Yan, Haokui Zhang, Guangda Huzhang, Yang Li, Yibo Wang, Qing-Guo Chen, Zhao Xu, Weihua Luo, Ying Li, Wei Dong, and Chunhua Shen. M^2: Dual-memory augmentation for long-horizon web agents via trajectory summarization and insight retrieval, 2026. URL https://arxiv.org/abs/2603.00503.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In *NeurIPS 2022*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/82ad13ec01f9fe44c01cb91814fd7b8c-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Peng Yuan, Yuyang Yin, Yuxuan Cai, and Zheng Wei. Webforge: Breaking the realism-reproducibility-scalability trilemma in browser agent benchmark, 2026. URL https://arxiv.org/abs/2604.10988.

Yuwen Zhai, Runze Li, Liang Wang, Nian Shi, Liwu Xu, Wei Zhang, Ran Lin, Bo Xu, and Benlei Cui. Guide: Interpretable gui agent evaluation via hierarchical diagnosis, 2026. URL https://arxiv.org/abs/2604.04399.

Jiayuan Zhang, Kaiquan Chen, Zhihao Lu, Enshen Zhou, Qian Yu, and Jing Zhang. Prune4web: Dom tree pruning programming for web agent. In *AAAI 2026*, 2025. URL https://arxiv.org/abs/2511.21398.

Yao Zhang, Shijie Tang, Zeyu Li, Zhen Han, and Volker Tresp. Webarbiter: A principle-guided reasoning process reward model for web agents. In *ICLR 2026*, 2026a. URL https://arxiv.org/abs/2601.21872.

Yuzhe Zhang, Xianwei Xue, Xingyong Wu, Mengke Chen, Chen Liu, Xinran He, Run Shao, Feiran Liu, Huanmin Xu, Qiutong Pan, and Haiwei Wang. Don't act blindly: Robust gui automation via action-effect verification and self-correction. In *ACL 2026*, 2026b. URL https://arxiv.org/abs/2604.05477.

Zirui Zhao, Jun Hao Liew, Yan Yang, Wenzhuo Yang, Ziyang Luo, Doyen Sahoo, Silvio Savarese, and Junnan Li. Gpa: Learning gui process automation from demonstrations, 2026. URL https://arxiv.org/abs/2604.01676.

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.

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

Shuyan Zhou. Webarena-infinity: Generating browser environments with verifiable tasks at scale. In *Blog Post*, 2026. URL https://webarena.dev/webarena-infinity/.

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

APPENDICES

The appendices provide implementation details, complete experimental protocols, additional comparisons, and prompt templates.

A IMPLEMENTATION AND METHOD DETAILS

This appendix specifies the synthesis pipeline, the applicability-verification loop, the tool and Tool-Use Contract schema, the online selection gate, and worked tool examples, expanding the main paper's Method.

A.1 SYNTHESIS PIPELINE

Each source task passes through four offline stages. (i) *Rollout*: a strong model runs the plain browser agent with an 18-step budget and records actions and screenshots. (ii) *Source assessment*: WebJudge scores the rollout. An end-to-end seed requires a successful trajectory with concrete extracted evidence; a failed rollout may contribute a locally useful segment after component-level checks. Thus, source assessment uses the task-level evaluator, while the generated applicability variants are checked without individual reference answers. (iii) *Synthesis and verification*: structural exploration produces atomic and composite tools, and applicability checks determine their admission. Origin labels such as `success_replay`, `local_fixed`, and `local_failed` support later filtering. (iv) *Freezing*: a content hash identifies the library snapshot used for reuse. The reported Claude Opus 4.6 snapshot contains 896 tools.

A.2 ACTIVE STRUCTURAL EXPLORATION

Structural probes inspect a control's enclosing container, repeated siblings, stable attributes (`data-testid`, `aria-label`, `name`, `id`, and role–name), and related pagination or sorting controls. Observed input options and URL parameters propose argument slots; network observations identify candidate read-only GET endpoints. The implementation stores locator alternatives in order. A fixed source-value execution checks the action pattern and evidence selector before parameterization. Selector repair receives the current observation and the failed locators.

A.3 APPLICABILITY VERIFICATION

The challenger proposes in-scope unseen values, out-of-scope intents, boundary values, and empty-result queries, retaining at most two usable inputs per class. Class counts vary with the generated list. Seed, unseen-value, and boundary probes use repeated execution, normally three trials; a majority of failed executions marks an unsuccessful probe. A further unseen-value or boundary execution is checked against the expected result type when it returns evidence. Query-invariant evidence, repeated content on an empty-result probe, explicit semantic rejection, or complete probe failure without a successful seed can demote a tool to `candidate`.

Empty-result probes screen for repeated page content by comparing nonempty outputs with successful-probe evidence; missing evidence produces no counterexample. An out-of-scope probe that returns evidence adds a rejection rule. Partial unseen-value failure restricts slots with verified values to those values; other slots retain their existing domains. The record distinguishes observed values from numeric intervals inferred between tested endpoints, and it does not treat a Cartesian product of per-slot domains as jointly verified. These updates operate per slot. Local repair allows two rounds by default and reruns the supplied challenges. Optional iterative challenging runs for at most three rounds, stopping on demotion, missing challenges, or no new counterexample class or rejection rule.

Admission also depends on the synthesis path. Parameterized local atoms use the challenger's demotion decision when tests are available; otherwise, repeated structural replay can admit them. An already-active composite retains its prior status when the optional challenge stage yields no tests. Parameterless extraction atoms combine structural replay with expected-result checking when a signature is present; fixed replay uses a lighter branch with separately configured semantic checking. Lifecycle records eligibility, while the generalization tier records the breadth of screened reuse.

Table 7: Tool and Tool-Use Contract fields.

Field	Meaning
`tool_id`	Unique identifier of the tool.
`impl`	Ordered action chain with selector fallbacks and {`slot`} templates.
`params`	List of parameters: `name`, `type`, `domain`/enum, `source` (task or page), `required`.
`robustness`	Verified-variant record from applicability verification (e.g. number of passing in-domain variants).
`contract.use_cond`	When the task and observation are in scope.
`contract.reject_rules`	Boundary of the applicability: neighboring intents just outside the tool's declared scope.
`contract.arg_reqs`	Required arguments that must be extracted.
`contract.expected_result_signature`	The self-predicted expected result type a valid result must match (its type, not its value).
`contract.description`	One-line description of the tool's action, used for retrieval.
`contract.lifecycle`	`candidate` or `active`.
`contract.risk`	`low` or `high`; gates automatic use.
`contract.gen_tier`	Generalization tier `fixed` or `paramed`; how far reuse is screened to reach.
`contract.role`	`standalone`, `atomic`, or `composite`.
`contract.provides/consumes`	Capability an atom offers, and upstream inputs it needs (e.g. `_page`).

A.4 Tool and Tool-Use Contract Schema

A synthesized tool is a `Tool` object with a `tool_id`, an implementation `impl`, a parameter list `params`, a `robustness` record, and a `contract`. Table 7 lists the fields. The implementation is a `SelectorChain`: an ordered list of action dicts, each with an op in {`goto_template`, `api_get`, `input`, `select`, `click`, `wait`, `extract`, `agent`, `call_tool`}, a selector-fallback list, and {`slot`} templates filled from arguments at execution. The stability preference among URL, backend-GET, UI, and agent implementations (labelled `L1` URL, `L2` backend-GET, `L3` UI, `L4` agent in the synthesis prompts) guides synthesis but is not stored on the contract, and the tool's bound site is derived from its first navigation rather than stored, so retrieval can hard-filter by host without a separate field. The ordering follows how much run-time state each form depends on: a URL template reaches the terminal page through the address bar without depending on DOM layout; a read-only backend `GET` is likewise layout-independent but needs a stable endpoint observed in network traffic; a semantic UI step depends on the family selector still resolving under the live DOM, so its behavior depends on the control remaining locatable; and a re-planned agent subgoal requires additional model decisions, so it is the last resort. Each step is realized at the highest form its probed evidence supports.

A composite tool invokes its components in the recorded order through `call_tool`, sharing one live browser session. Page-dependent atoms declare `consumes=[_page]` and are excluded from standalone retrieval, while a preceding component establishes their page state. The executor permits at most two nested call levels and rejects deeper calls.

A.5 Online Selection and the Gate

Retrieval precedes selection. It filters on active lifecycle, host, risk permission, and page prerequisites, then ranks candidates using description relevance, tier, and argument availability. The weak model proposes one tool and an argument dictionary at temperature 0. The gate checks $L \wedge H \wedge R \wedge J \wedge A$: lifecycle, host, risk, intent rejection rules, and arguments. The first, second, third, and fifth checks are programmatic. The intent check uses the selector model when rejection rules are present. Rejected proposals return to the browser agent; accepted proposals proceed to execution. Failure categories distinguish retrieval, use-condition, argument, execution, and evidence errors.

The gate short-circuits in order L, H, R, J, A. Lifecycle, host, or risk rejection avoids the intent-model call. The selector proposes one tool; selection, intent screening, extraction fallback, and agent

Table 8: Representative synthesized tools across implementation forms (URL template, backend GET, semantic UI); this stability preference is distinct from the contract's `fixed`/`paramed` generalization tier.

Form	Tool	Parameters	Implementation and evidence
URL template	`craigslist_apartment_filter`	min_bedrooms, min_bathrooms, max_price, price_period	`goto_template` into listing search; `extract` listing titles.
Backend GET	`ryanair_cheap_flights_search`	origin, destination, max_price, departure_date, airline	read-only `api_get` to fare-finder endpoint; parse fare list.
Semantic UI	Semantic UI form tool	task-dependent form fields	`input`/`select`/`click` with role–name fallback; `extract` terminal evidence.

subgoals use the deployment model. The original runs accept unparseable intent and expected-result judgments; the shared-budget experiments reject them.

The J check evaluates explicit rejection rules, with positive task matching performed during retrieval and selection. Host matching derives the bound site from navigation or a referenced component. Risk admission follows the runtime write permission. Each experiment freezes these options alongside its model, task, and library configuration.

The argument predicate A requires the arguments named by `contract.arg_reqs` to be present and nonempty. For a declared slot with a nonempty domain, it accepts exact membership or, when the stored values are numeric, min–max interpolation. Empty domains impose no value constraint, and undeclared extra arguments are skipped. Checks apply per slot, without validating the joint tuple. Numeric-shaped identifiers such as postal codes can therefore be treated as intervals even though their semantics are categorical.

A.6 Representative Synthesized Tools

Table 8 gives concrete tools across implementations. The URL-based `craigslist_apartment_filter` maps {`min_bedrooms, min_bathrooms, max_price, price_period`} into a listing-search URL and extracts listing titles as evidence; the backend-GET `ryanair_cheap_flights_search` issues a read-only `api_get` to a fare-finder endpoint templated on origin, date, and price; a UI-step tool performs semantic interactions with role–name fallback where form entry is required. These examples show how argument filling invokes URL, backend-API, and UI procedures.

To make the full Tool-Use Contract concrete, three `active` tools from the frozen Claude Opus 4.6 library illustrate representative contract fields. **(A1)** `gov.uk__nav_p0_verb` answers *"find the eligibility to get child benefit, how it works and how to claim,"* with `p0`∈{`child-benefit`} as its verified domain, an implementation that navigates `goto_template https://www.gov.uk/{p0}` and extracts, and an expected result type of "a summary of eligibility criteria, how the benefit works, and how to claim it." **(A2)** `apple.com__nav_p0_verb` answers *"find technical specs for the latest MacBook Air on Apple"* with a two-slot path template (`p0`∈{`macbook-air`}, a second slot ∈{`specs`}) and an expected result type of "a list of technical specifications (processor, RAM, storage, display, battery)." **(A3)** `coursera.org__nav_p0_verb` answers *"search for a beginner computer-science course that includes Python"* with a query-plus-faceted-filter template (`p0`∈{`search`}, a `query` slot, and a difficulty facet). **(A4)** The same Apple contract also records its rejection edge: a `reject_rules` entry marks a look-alike cross-brand intent ("find technical specs for the latest Dell XPS 13 on Dell's website") as out of scope, providing an explicit boundary for intent screening. Each contract field, the verified domain, the expected result type, and the rejection edge, is the object the applicability challenger checks during synthesis.

Table 9: Representative passing tool-use cases retained from the earlier deployment logs. The weak agent reads the contract and fills arguments; a composite can package multiple internal actions into one agent-visible call. These examples are separate from the main repeat-run aggregates.

Task	Tool	Arguments filled by the weak model	WebJudge
Top posts from last year on 9GAG	`9gag.com__nav_p0_verb`	{`p0:top, top:year`}	pass
Beginner CS course with Python on Coursera	`coursera.org__nav_p0_verb`	{`p0:search, query:beginner ...python`}	pass
Daily weather forecast for a city	`accuweather.com__nav_daily-weather-forecast`	{`id:349727`}	pass
ID needed for an Amtrak trip	`amtrak_com__composite`	{} (fixed composite)	pass
Most Popular TV on Rotten Tomatoes	`rottentomatoes_com__composite__fixed`	{}	pass
Filtered bluetooth wireless speakers on Best Buy	`bestbuy_com__task_0__fixed`	{}	pass

Table 10: Frozen Claude Opus 4.6-library inventory and usage flags in the selected main run. Inventory counts describe the stored tool objects; `tool_used` is a recorded outcome flag rather than a complete execution trace.

Quantity	Value
Source tasks	190
Total tools	896
active / candidate	236 / 660
fixed / paramed	787 / 109
standalone / atomic / composite	230 / 490 / 176
Sites with at least one active tool	68
Evaluated tasks	159
Flagged uses	35
Successes among flagged uses	19

A.7 Library Inventory and Firing Rates

The frozen Claude Opus 4.6 library contains 896 tools from 190 source tasks (mean 4.72 tools per source task, maximum 16). Of these, 236 are `active` and 660 are `candidate`; 787 carry the `fixed` tier and 109 the `paramed` tier. Roles comprise 230 standalone, 490 atomic, and 176 composite tools. The library inventory includes 88 sites with synthesized tools and 68 with active tools. The main repeat records contain 159 task IDs across 68 recorded site strings. In the selected main run, the `tool_used` flag is true on 35 tasks, with 19 successful tasks in this set. Table 10 separates the static inventory from these deployment observations.

A.8 Failure Taxonomy and Maintenance

Failure records identify where maintenance may be needed. Table 11 maps observed signals to likely causes and targeted follow-up actions. A failed selector suggests updating the implementation; an argument mismatch suggests revisiting the domain or extraction; and a misleading tool description suggests revising its use conditions. These records support maintenance after diagnosis rather than guaranteeing automatic correction of every failure.

Table 11: Failure signals and maintenance actions.

Failure signal	Likely cause	Maintenance action
Tool not fired, fallback succeeds	Coverage or selection gap	Widen use conditions or synthesize a task-matched tool.
Wrong tool selected	Cross-site or contract mismatch	Cross-site guard; add rejection rules or split the contract.
Required argument missing	Argument-extraction failure	Improve the extractor or require fallback.
Tool fired, evidence invalid	Selector or endpoint drift	Update the implementation; re-verify.
No tool synthesizable	Dynamic widget / free-text / aggregation	Report as an abstraction-boundary case.

A.9 Case Studies: Tool Reuse and Contract Boundaries

The following cases retain examples from the earlier deployment records. They illustrate how navigation, extraction, and rejection rules appear in concrete tools. Their individual step counts refer to those case runs, rather than the selected run in Table 1.

9GAG (composite, one step vs. a full budget). For the task *"Find the top posts from last year on 9GAG"* the weak no-tool agent clicks around the home page for the full 30-step budget and never finds the year view, scoring a failure. The synthesized composite tool `9gag.com__nav_p0_verb` takes {`p0=top, top=year`}, navigates `goto_template 9gag.com/{p0}/{top}`, calls its in-page atom, and returns the ranked list in a single agent-level tool call. Its expected result type is fixed at synthesis as "a list of top posts over the past year with engagement metrics," to guide the evidence check. In this recorded case, $0 \rightarrow 1$ success, and the agent-visible step count falls from the full 30-step budget to the one tool call ($30 \rightarrow 1$); the composite's internal browser primitives (the `goto_template` and the in-page `call_tool`) execute inside that one call and are not separate agent steps. Figure 3 visualizes the complete source-to-tool-to-reuse path.

The figure reads left to right in three stages. Stage 1 is the single recorded path the tool is built from: the agent moves through the home page, the Top view, and the Last Year control to a ranked list. Stage 2 is synthesis: the pipeline recovers that path, binds its arguments as {`p0=top, top=year`}, and fixes the terminal check the tool must satisfy (a non-empty ranked list), emitting the composite `9gag.com__nav_p0_verb` with implementation `goto_template 9gag.com/{p0}/{top}` chained to its in-page atom. Stage 3 places the two runs side by side on the same task: without the tool the weak agent explores the home page for the full step budget and fails, while with the tool one call lands on the ranked-list result and passes WebJudge. The screenshots in Stages 1 and 3 are the recorded home-page and ranked-list pages for this task, so the figure shows the same $0 \rightarrow 1$ success and $30 \rightarrow 1$ step change reported above, traced from source trajectory through synthesis to weak-model reuse.

NBA (composite with intent rejection rules). For *"how many assists is Chris Paul averaging this season"*, `nba_com__composite` navigates to the player page, calls its extract atom, and checks for a non-empty numeric result. Its expected result type is "a numeric assists-per-game statistic." The stored `reject_rules` exclude *"how many assists Chris Paul averaged in the 2018–2019 season"* and *"how many assists LeBron James has been averaging"*. These rules specify a different season and player, respectively, and the tool records `verified_variants`=2 from the challenger. At runtime, the intent predicate J evaluates requests against these rules; positive task matching occurs during retrieval and selection.

Craigslist (URL template and argument domain). A listing-filter tool maps {`query, srchType, max_price, condition`} into a search URL and extracts listing titles. The `condition` slot carries enumerated values, allowing the argument check to accept members and reject other values. The example illustrates how parameter scope is exposed in a callable interface; testing broader values requires executing the intended arguments and evaluating the resulting task outcome.

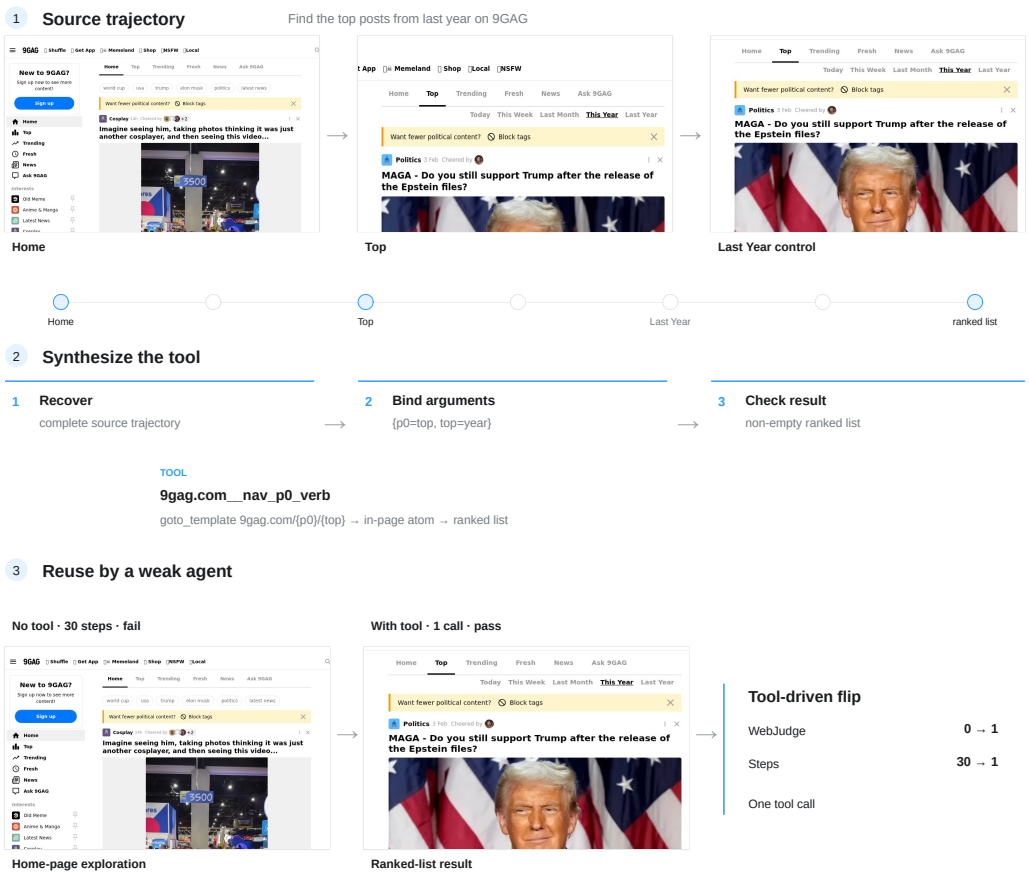


Figure 3: *End-to-end 9GAG case study.* From the source trajectory, TrajSmith synthesizes the composite tool `9gag.com__nav_p0_verb`. For the task "Find the top posts from last year on 9GAG," the weak Qwen3-4B agent fills {`p0=top, top=year`}; the composite navigates to `9gag.com/{p0}/{top}`, calls its in-page atom, and returns a non-empty ranked list. The no-tool run exhausts the 30-step budget and fails, whereas the with-tool run passes WebJudge in one call (success $0 \to 1$; steps $30 \to 1$).

A.10 Ablation Settings

The library-view controls operate on the frozen library: candidate-inclusive retrieval also admits `candidate` tools, and failed-origin removal excludes tools tagged `origin:local_fixed`. Runtime controls remove retrieval ranking, semantic checking, or model-assisted extraction. The extraction ablation disables the online executor's text-based model fallback for DOM extraction; the normal online setting uses the deployment model. Table 5 uses 58 observed tasks for every condition. Separately rebuilt libraries are reported in Appendix C.9.

B Live Benchmarks and Reachability

B.1 Benchmark Task Sets

The Online-Mind2Web collection records 147 site entries and 98 reachable entries, corresponding to a 194-task pool. Blocking, unavailable pages, and unstable interaction account for access failures. The initial collection executes 191 tasks and retains three further tasks in its pool accounting. The main repeats use the archived 159-task subset selected for tool coverage, omitting the other 35 pool tasks from both arms. Each repeat reports the observed 159-task denominator rather than substituting outcomes for the unexecuted tasks.

Table 12: Source/evaluation overlap for the Claude Opus 4.6 library and the main 159-task repeat evaluation.

Quantity	Value
Main repeat evaluation tasks	159
Also used as synthesis source	155
Absent from recorded synthesis sources	4
Distinct source tasks in the library	190
Tools per source task (min/mean/max)	1 / 4.72 / 16

The WebArena evaluation retains 763 single-domain tasks after excluding 48 cross-site tasks and GitLab task 411. Both conditions use Qwen3-4B-Instruct-2507 and the native task checker; the tool condition adds a frozen library constructed by GPT-5.4.

C Evaluation Protocol and Analyses

C.1 WebJudge

We adopt the official WebJudge protocol of Online-Mind2Web (Xue et al., 2025) without modifying the vendored evaluation code. WebJudge runs in three stages: it identifies task key points from the instruction, scores each recorded screenshot for relevance on a 1–5 scale and retains those at or above a threshold of 3 (capped to bound token cost), and issues a final `success`/`failure` verdict over the retained screenshots and action thoughts. We record each run as a screenshot-and-action trajectory in the official directory format, including a final-state screenshot, so trajectories are judge-ready.

The original runs use Claude Opus 4.6 as WebJudge, separately from its synthesis and source-rollout roles. The additional shared-budget batches use GPT-4.1 as WebJudge (Appendix C.12). A retained cross-model check on the original trajectories is described below.

Crash handling. Main-table success counts include crashed runs as failures. Repeat 1 contains two no-tool crashes and repeat 2 contains one, with no tool-arm crashes in either repeat. Excluding every task pair containing a crash gives $29 \rightarrow 48$ successes on 157 tasks and $28 \rightarrow 47$ on 158 tasks, respectively. The complete-pair analysis retains all 159 tasks in each repeat.

C.2 Source Tasks and Evaluation Tasks

The frozen Claude Opus 4.6 library has 190 unique source IDs. Table 12 gives the intersection with the 159-task repeat evaluation: 155 shared IDs and four other IDs. The additional experiments specify their construction and evaluation sets separately.

C.3 Applicability-Verification Records

The following records describe candidate admission and encountered counterexamples. Admission counts are the screen's decisions; task success is evaluated separately.

Admission audit. The original diagnostic sample contains 100 tools with logged admission decisions: 78 become active and 22 remain candidate. Of the admitted tools, 24 passed the initial semantic check and 54 were recorded as admitted after repair. A related challenge log covers 104 tools and contains 15 `query_invariant`, 15 `empty_recommend`, 15 `verifier_reject`, and one `total_fail` event. A tool can trigger multiple event types, so event totals are not a confusion matrix. Table 13 preserves the dispositions separately from deployment success.

C.4 Cross-Model Task Evaluation

Cross-model judge check. A separate diagnostic rescored 33 historical strong-selector tool trajectories with GPT-4.1 under the WebJudge protocol. The judges agree on 27: 22 passes and five

Table 13: Applicability-pass dispositions in the 100-tool decision sample. These are logged decisions, without independent correctness labels.

Applicability-pass outcome	Count
Admitted active	78
Retained candidate	22
Admitted after initial semantic check	24
Recorded as repaired then admitted	54

Table 14: GPT-4.1 reuse on 151 observed Online-Mind2Web task pairs. GPT-4.1 supplies synthesis and online execution; Claude Opus 4.6 supplies WebJudge. All reported metrics use the same task set.

Method	Solved tasks	Success (%)	Mean steps
No-tool	27	17.9	21.7
TrajSmith	35	23.2	14.4

failures. There are three Claude Opus 4.6-only passes and three GPT-4.1-only passes. Among the 25 Claude Opus 4.6 passes, three are disputed by GPT-4.1. This check measures agreement between task evaluators.

C.5 Reuse with GPT-4.1 as Synthesizer and Online Agent

Table 14 reports reuse with GPT-4.1 as both synthesizer and online agent on 151 observed task pairs. Tools improve success from $27/151$ to $35/151$ (17.9% to 23.2%). Both conditions use Claude Opus 4.6 WebJudge, and their step means use these same 151 tasks. This diagnostic complements the weak-agent comparisons by testing a second construction model.

C.6 Firing and Paired Outcomes

The selected 159-task run has paired outcome cells $(\text{both pass}, \text{no-tool only}, \text{tool only}, \text{both fail}) = (19, 10, 30, 100)$, giving $29 \to 49$ successes. The additional run has cells $(22, 6, 25, 106)$ and $28 \to 47$ successes. Each uses the same 159 task IDs in both arms, with no duplicate IDs. The main table selects the run with higher TrajSmith success; these are single-run results, not averages. In the additional run, easy, medium, and hard counts change from $12 \to 24$, $11 \to 19$, and $5 \to 4$, respectively.

Mean recorded agent-loop steps are $17.006 \to 14.484$ and $17.566 \to 14.447$. The averages include unsuccessful runs. A tool call may contain navigation, extraction, nested calls, or agent subgoals, so an agent-visible step is not a primitive browser action. Success-conditioned traces help interpret the mean, but step counts alone do not establish wall-clock or total-compute savings.

C.7 Attributing the Gain: Firing, Assisted, and Fallback

We distinguish a tool attempt, accepted tool evidence, and final task success. An attempted tool may change the browser state before fallback; an abstaining run continues with the browser policy. Outcome variation can also arise from model decisions and changing live pages.

The selected main run records `tool_used=true` on 35 tasks, with 19 successes in this group. Among the 30 positive task-outcome changes, 16 carry this flag. Table 15 summarizes these associations. The flag records accepted use; earlier execution followed by fallback may leave it false.

What the gate rejects. The retained decision-level diagnostic contains 128 logged decisions, including 65 tool proposals and 63 abstentions. Replaying programmatic checks rejects three proposals on host and three on arguments, leaving 59. The model-based intent check is not rerun in this analysis. These counts describe logged decisions, not distinct task successes or the usage frequency in the main repeats.

Table 15: Usage flags and paired positive changes in the selected 159-task run. Flagged-use associations are not estimates of a separate causal tool effect.

Recorded quantity	Tasks
Evaluated tasks	159
`tool_used=true`	35
Success among flagged uses	19
No-tool fail to tool-arm pass	30
Flagged uses among these positive changes	16

Table 16: Historical decision-level gate diagnostic. Programmatic checks are replayed; the model-based intent check is not. Counts describe decisions rather than distinct deployment tasks.

Gate stage	Count
Decisions logged	128
model picked a tool	65
model abstained (no pick)	63
Pre-gate candidates (mean / max)	8.02 / 12
Rejections among picked (exact predicates):	
L lifecycle (not active)	0
H host (site mismatch)	3
R risk (write, no permission)	0
A args/domain (missing or out-of-domain)	3
passed all exact predicates	59

Table 17: Recorded efficiency counters for the selected 159-task run. Steps count outer agent-loop interactions; the call counter does not include all possible tool-internal calls.

Method	Mean steps	Recorded LLM calls
No-tool	17.006	17.006
TrajSmith	14.484	15.138

Step efficiency. Table 17 gives step and call counters from the same selected run as the main results. Agent-loop calls can decrease when a tool directly returns evidence, but the historical aggregate call counter omits some internal executor calls. Complete cost accounting therefore requires both outer-loop and tool-internal traces.

C.8 Statistical Significance

We compute a two-sided exact McNemar test from the discordant counts b (no-tool pass, tool fail) and c (no-tool fail, tool pass): $p = \min\{1, 2\sum_{k=0}^{\min(b,c)} \binom{b+c}{k} 2^{-(b+c)}\}$. The selected 159-task run has $b = 10$, $c = 30$, and $p = 0.00222$; the additional run has $b = 6$, $c = 25$, and $p = 0.000878$. These values describe within-run paired outcomes. They are nominal tests without adjustment for selecting the higher-scoring run.

Repeated task IDs, the strong-selector diagnostic, and the selected 58-task comparison remain separate analysis units.

C.9 Separately Synthesized Internal Controls

The internal controls rebuild libraries from the source collection while disabling exploration and applicability verification. They differ from ablation views of the existing library. Table 18 compares the full and passive/no-verification libraries on the same 58 tasks, retaining the shared no-tool reference. Both libraries solve 18 tasks. Realized construction cost is not matched in these records.

Table 18: Separately synthesized libraries on 58 shared tasks with Qwen3-4B. All rows use the same task IDs; construction cost is not matched.

Method	Solved tasks	Success (%)
No-tool	5	8.6
Full TrajSmith	18	31.0
Passive / no verification	18	31.0

Table 19: Usage diagnostics for weak-selector ablations on 58 shared tasks. Accepted-use success is conditional on the recorded `tool_used` flag.

Configuration	Flagged uses	Successes among uses
Full	18	8/18
w/o retrieval	18	7/18
Include candidate tools	17	9/17

C.10 Component Ablation Under the Deployment Condition

Table 5 reports weak-selector ablations on the common 58-task set. The reused no-tool reference solves five tasks and uses the historical 30-step configuration; tool arms use the 18-step fallback path with routing. Among the tool arms, retrieval removal has the largest observed loss. Removing semantic verification or failed-origin tools improves the count by one and two, respectively.

Table 19 adds usage information for the same 58-task set. Full and no-retrieval each record 18 accepted uses, with eight and seven successes among those uses. Candidate-inclusive retrieval records 17 accepted uses, with nine successes. These counts describe success conditional on accepted tool use.

C.11 Comparison with Tool-Learning Baselines

The shared 58-task baseline set contains five weak no-tool successes and 53 failures. ContractSkill and TrajSmith are evaluated on these same IDs with Qwen3-4B downstream and Claude Opus 4.6 WebJudge. Table 4 reports five, six, and 18 successes for no-tool, ContractSkill, and TrajSmith. Selection using source outcomes makes this a targeted comparison subset rather than a random sample of the benchmark.

The ContractSkill adapter generates a contracted skill from the task and current page, executes it through Playwright, checks its postcondition, and repairs failed skills. The $6/58$ condition uses Qwen3-4B for this per-task generation and repair loop. TrajSmith supplies a frozen library constructed offline by Claude Opus 4.6. The original no-tool reference allows 30 steps; TrajSmith uses one routing step and up to 18 fallback steps.

C.12 Shared-Budget Deployment Protocol and Source Overlap

The additional experiments examine reuse with smaller deployment models under shared call and action caps. The first batch evaluates 62 tasks on 61 sites with the frozen 896-tool Claude Opus 4.6 library; 60 task IDs overlap its sources. The initial 64-task list excludes two instructions requiring external contact or a vehicle-service submission, by task semantics across all arms. The subsequent weak-model comparison takes the first 32 eligible tasks from that list, selected before its runs, and evaluates both GPT-4o-mini and Qwen3-VL-4B. The batches share task IDs and are not pooled as independent observations.

All online model calls use the specified deployment model, including selection, intent checks, tool-internal model calls, and fallback. The shared budget counts at most 18 attempted calls, including retries, and 30 BrowserPort action units. A type-and-submit action costs two units; initial navigation is recorded separately in every arm. The 300-second deadline is checked between requests and actions, so an in-flight operation may finish later. A worker has a 540-second total timeout includ-

Table 20: Mean per-task online work for Qwen3-VL-4B on the same 32 tasks as Table 3. Calls include internal requests and retries; actions use the BrowserPort unit definition above. Times include agent-stage initialization and recording and depend on concurrency.

Method	Calls	Actions	Input tokens	Seconds
No-tool	15.63	16.50	55,984	94.9
TrajSmith	14.22	15.25	45,581	86.3

ing evaluation. WebJudge uses GPT-4.1 separately from this online budget. Failures and missing verdicts count as failures. The 62-task full and candidate-inclusive arms each contain one browser-startup failure without a verdict; recovery runs under new IDs leave the aggregate success counts unchanged. The 32-task batches have complete verdicts and no worker failures.

The 32-task batches enable the same action-target normalization in all arms: observed element IDs and role/name targets are converted to the browser interface using the current observation. The runner also saves partial progress before a budget exception and logs tool attempts separately from accepted evidence. Each configuration has its own frozen records.

Broader task-set check. On the 62-task set, GPT-4o-mini solves 18 tasks without tools (29.0%) and 25 with the active library (40.3%). Candidate-inclusive retrieval solves 27 (43.5%); it admits candidate entries from the same frozen library while retaining the other gates. This comparison changes retrieval eligibility, not the synthesis procedure.

Paired uncertainty and attribution. For Qwen3-VL-4B on the 32-task set, the paired outcome cells (both pass, no-tool only, full only, both fail) are $(8, 0, 4, 20)$, with two-sided exact McNemar $p = .125$.

Qwen3-VL-4B attempts a tool on 12 tasks: 11 executions return and one is interrupted. One task accepts tool evidence, ten executions return to fallback, and 20 episodes abstain before execution. Tool attempts can change the page state and consume budget before fallback, while abstaining episodes still vary with model output and live-page state. The success differences therefore describe the complete configurations, without attributing every additional success to accepted tool evidence.

Model identities and execution records. The shared-budget API runs use GPT-4o-mini-2024-07-18 for deployment and GPT-4.1-2025-04-14 for WebJudge, as identified in the responses. Qwen3-VL-4B-Instruct runs locally in bfloat16 with Transformers 4.57.3, PyTorch 2.6.0, greedy decoding, a 20,000-token input limit, and at most 2,048 generated tokens per request on one H200. No model is fine-tuned. Qwen3-4B-Instruct-2507 is the text-only deployment checkpoint; Qwen3-VL-4B-Instruct is the separate multimodal checkpoint used in the shared-budget runs. Each episode saves its task and configuration, model requests, browser actions, screenshots, final state, and judge verdict; frozen-library hashes are checked against the manifests. Shared caps do not equalize tokens, dollar cost, or latency.

C.13 Repeated Public-Task Evaluation

A further GPT-4o-mini run uses the same 32-task Online-Mind2Web subset. The runner verifies each task ID, instruction, and starting URL against the archived dataset. Both conditions use the same browser implementation, call and action caps, and GPT-4.1 evaluator. The existing library remains frozen throughout evaluation.

The selected GPT-4o-mini run solves $9/32$ tasks without tools and $16/32$ with TrajSmith; the other completed run solves $9/32$ and $15/32$. In the selected run, the paired cells (both pass, no-tool only, full only, both fail) are $(9, 0, 7, 16)$, giving nominal $p = .015625$ without adjustment for run selection. Of the seven gains, one accepts tool evidence, two follow tool execution and fallback, three abstain before execution, and one has an interrupted tool attempt but a successful final-page verdict. Each batch is analyzed separately; the success difference measures the complete configuration, including selection and fallback.

D Scope and Future Directions

TrajSmith works with recurring, site-specific procedures whose inputs and outputs can be identified from page observations. The current compositions follow recorded action order. Dynamic widgets, branching procedures, cross-site tasks, and cross-item aggregation require richer observation and composition. The hard-task breakdown motivates this extension.

Expected-result checking tests observable output properties. It can miss incorrect values or unmet task constraints. Domain representations also need to distinguish quantities, identifiers, free text, and joint argument conditions; numeric interpolation can admit untested values. Independently labeled execution outputs would support a direct assessment of verification precision.

The main live-web evaluation measures reuse on mostly previously attempted tasks. Broader reuse requires recovering useful intermediate states, exposing the parameters that actually change behavior, and matching those procedures to new requests. Live-site access and model variation continue to affect repeatability.

E Reproducibility and Responsible Research

E.1 Computational Reporting

No model is trained or fine-tuned. Claude Opus 4.6, accessed through the API identifier `aws.claude-opus-4.6`, supplies the original Online-Mind2Web source rollouts, synthesis, applicability checks, and WebJudge evaluations. The 159-task repeats deploy Qwen3-4B-Instruct-2507 through a local vLLM endpoint; selection, model-assisted execution, and fallback use this deployment model. The recorded one-time library-construction cost is $41.25. WebArena uses GPT-5.4 offline and Qwen3-4B-Instruct-2507 online, with the native deterministic checker. The shared-budget comparisons retain the Claude Opus 4.6 library and use GPT-4.1 for evaluation. Appendix C.12 gives their deployment-model versions, token counts, and runtime settings. Evaluation calls are accounted for separately.

Source rollouts use an 18-step budget in the original collection, and model observations resize screenshots to a maximum edge of 1024 pixels. The paired live-web repeats allow 18 no-tool steps and one routing step followed by up to 18 fallback steps. WebArena allows 30 steps. Shared-budget experiments count model-call attempts and browser action units directly across the complete online path.

E.2 Artifacts, Privacy, and Safety

The research artifacts include synthesized tools, contracts, task manifests, evaluation summaries, and execution evidence. A release should retain the configurations needed to interpret each experiment while excluding credentials, private account state, and site content that cannot be redistributed. Upstream benchmark and software terms apply. The artifact package should also distinguish the frozen evaluated implementation from later development changes. AI assistance is disclosed in the main paper's AI Use Statement.

F Ethical Statement

TrajSmith aims to reduce repeated browser planning through reusable procedures and inspectable use conditions. Such procedures can also accelerate unwanted automation. Deployments should respect website access controls and rate limits and obtain authorization for state-changing actions. Self-hosted benchmark state isolates these actions during evaluation.

Execution logs can contain account state or third-party content. Artifact release requires credential removal, privacy review, and compliance with benchmark and website redistribution terms. Consequential applications require task-specific validation and human review. Demographic fairness is outside this evaluation; applications affecting people need a separate assessment of model, content, and coverage biases.

G PROMPT TEMPLATES

The following templates specify the synthesis, verification, and deployment prompts. Template fields are filled programmatically, and JSON requests specify the output format. Instructions in a prompt express intended behavior; actual enforcement depends on the parser, executor, and configuration described in Appendix A.

G.1 S1: SUBTASK DECOMPOSITION

Identifies a reusable subtask, interaction mode, and candidate parameter slots.

```
Given a web TASK, identify the SINGLE reusable subtask needed to complete it,
its interaction MODE (one of: search / filter-locate / multi-step-filter /
availability-check / location-lookup / form-submit), and every VARIABLE
parameter slot (not just the main query -- also things like hours_min,
sentiment, sort, date, location). Prefer deterministic steps (stable selectors
or query-URL templates); an 'agent' step is only for a small uncertain goal.
Return ONLY JSON: {"subtask":str,"mode":str,
"params":[{"name":str,"type":"str","source":"task"}]}

TASK: {task}
```

G.2 S2: STRUCTURAL EXPLORATION

Requests live probes and an executable proposal, preferring supported direct navigation or read-only endpoints before UI actions and agent subgoals.

```
You are exploring a real website to build a reusable, parameterized tool for
the given TASK and its SUBTASK/MODE. Choose the SIMPLEST tier that achieves the
subtask, and PROPOSE it. Try tiers IN ORDER:
  (L1) DIRECT query-URL: if filters/query live in the URL, goto it to CONFIRM
       results, then propose ONE goto_template + ONE extract.
  (L2) READ-ONLY GET API: if the page loads results from a GET JSON endpoint
       (check the CAPTURED APIS list), propose an api_get hitting that endpoint
       with mapped params -- the most robust when available.
  (L3) UI STEPS: otherwise reproduce the minimal input/select/click/wait steps.
       Prefer SEMANTIC selectors (data-testid/aria-label/name/id or role+name),
       NEVER bind to a bare positional index like :nth-of-type(3).
  (L4) AGENT escape hatch: only for a small uncertain sub-goal.
Exploration actions (probe the live page) -- return ONE JSON:
  {"op":"goto","url":...} | {"op":"type","sel":[css...],"text":...,"submit":true}
  | {"op":"select",...} | {"op":"click",...} | {"op":"giveup"} | {"op":"propose",...}

PROPOSE the reusable tool with an ordered STEPS list and a TIER label.
Parameters as {slot}; parameterize EVERY variable slot; keep steps DETERMINISTIC:
  {"op":"propose","tool_id":str,"tier":"L1|L2|L3|L4",
   "steps":[
     {"op":"api_get","url_template":"...{slot}...","json_path":"data.results","as":"key"},
     {"op":"goto_template","url_template":"...{slot}..."},
     {"op":"input","sel":[css...],"text":"{slot}","submit":true,"role":"textbox","name":".
     ↪  .."},
     {"op":"select","sel":[css...],"value":"{slot}","role":"combobox","name":"..."},
     {"op":"click","sel":[css...],"role":"link","name":"..."},
     {"op":"wait"},
     {"op":"extract","sel":[css...],"as":"key","pick":"first"},
     {"op":"agent","goal":"...{slot}...","max_steps":3}
   ],
   "params":[{"name":..,"type":"str","source":"task","required":true}],
   "evidence_sel":{"titles":[css...]},  // MUST select the actual RESULT ITEMS
     (list rows / product cards / result links / the answer value), NOT the page
     heading/title/result-count; NEVER bare 'h1'/'h2' (a count header like
     'Showing 1-50 of 2521' is NOT valid evidence); NEVER catch-all
     'body'/'pre'/'html'/'*'/bare 'div'/'span' (they match the page shell and
     falsely pass even when NO results loaded).
   "example_args":{"<param>":"<REAL value you used, NOT 'test'>"},
   "use_cond":"<one-line WHEN-to-use description, for reuse retrieval>"}

Every required param MUST appear as {name} in some step; do NOT declare a param
you don't wire in. If the TASK's key dimension cannot be wired into THIS tool,
build a SEPARATE fine-grained tool rather than a one-size catch-all.
Do NOT repeat a HISTORY action that did not change the OBSERVATION -- switch tier
or giveup. As soon as a tier is CONFIRMED working, PROPOSE immediately.

TASK: {task}   SITE: {site}   HISTORY: {hist}
```

```
CAPTURED APIS (GET JSON endpoints seen while exploring): {apis}
OBSERVATION: {obs}
```

G.3 S3: EVIDENCE-SELECTOR CORRECTION

Requests a replacement when proposed evidence selectors match no result content.

```
You proposed a reusable tool but its evidence selectors matched NOTHING on the
current result page. You ARE on the correct result page. Look at the OBSERVATION
and pick CSS selectors that WILL match the actual RESULT ITEMS/content -- NOT the
page heading/title/result-count. NEVER return bare 'h1'/'h2', and NEVER use
catch-all 'body'/'pre'/'html'/'*'/bare 'div'/'span'. Prefer semantic classes,
list-item tags, aria roles. Return ONLY JSON: {"evidence_sel":{"titles":[css...]}}

TASK: {task}   PREVIOUS (missed) evidence_sel: {prev}   OBSERVATION: {obs}
```

G.4 S4: PARAMETER PROBING

Identifies variable parts of a recorded action from its DOM context in the successful-trajectory branch.

```
You are analyzing one step of a successful web task trajectory to make it
reusable. Given the DOM context and the action, propose which parts of this
step are variable parameters (query text, category, price direction, star
rating, address fields, page/sort, etc.). Return ONLY JSON:
{"param_slots":[{"name":..., "type":..., "source":"task|page"}]}

DOM: {dom}   ACTION: {action}
```

G.5 S5: CONTRACT DRAFTING

Produces use conditions, rejection rules, and an expected-evidence description for the successful-trajectory branch; failed-rollout tools are described from their recovered page state.

```
You are drafting a Tool-Use Contract for a reusable web tool distilled from a
successful trajectory. Given the task and the per-step param slots, produce a
contract. Return ONLY JSON with keys: use_cond (string), reject_rules (list of
strings describing incompatible requests to refuse), result_evidence (string).

TASK: {task}   PARAM_SLOTS: {slots}
```

G.6 S6: APPLICABILITY CHALLENGER

Proposes alternatives, empty-result requests, neighboring intents, and boundary cases from the contract. Here, "independent challenger" denotes a separate prompt role; the original synthesis and challenge calls use the same Claude Opus 4.6 model.

```
You are an independent CHALLENGER auditing a reusable web tool's GENERALIZED
capability claim. You are given ONLY the tool's public CONTRACT (its declared
purpose and parameters), NOT its implementation or the demonstration that
produced it. The tool was admitted because ONE source task succeeded, but that
does NOT prove the capability holds across its declared parameter neighborhood.
For EACH parameter, produce alternative VALUES a real user might supply within
the SAME subtask, plus values that look valid but should yield NO results, plus
NEIGHBORING intents OUTSIDE this tool's scope, plus a boundary/degenerate but
still-valid value. If a parameter lists known_values, match their EXACT FORMAT
(e.g. known_values=['la-liga'] -> 'bundesliga', not 'Bundesliga'). Do NOT invent
selectors, URLs, form fields, or API details, only parameter VALUES and intents.
Return ONLY JSON: {
  "in_domain_unseen":   [{<param>: <valid value, not the seed>}, ...],
  "empty_result_valid": [{<param>: <format-valid value expected to return NOTHING>}, ...],
  "out_of_scope":       [{"task": <neighboring intent to REFUSE>, <param>: <value>}, ...],
  "boundary_form":      [{<param>: <boundary/degenerate but valid value>}, ...] }

TOOL PURPOSE (use_cond): {use_cond}
EXPECTED RESULT: {result_evidence}   PARAMETERS: {params}
```

G.7 S7: RESULT-SIGNATURE JUDGE

Compares extracted evidence with a generated output description, without a per-variant reference answer.

```
A reusable web tool declared, at synthesis time, that a successful run should
return a result matching this SIGNATURE (a description of the expected kind of
answer, predicted by the tool author, NOT the ground-truth answer). Judge whether
the ACTUAL EXTRACTED OUTPUT matches that signature: it must be a concrete result
of the described kind (e.g. a specific numeric value, a matching listing, the
requested data point), NOT page chrome, navigation headers, section titles, or
generic links. Return ONLY JSON: {"matches": true|false, "reason": "<short>"}.

EXPECTED RESULT SIGNATURE: {signature}   ACTUAL EXTRACTED OUTPUT: {evidence}
```

G.8 O1: TOOL SELECTION

Requests one tool and its arguments, or abstention, from a catalog of public tool fields.

```
You are a small web agent that can call pre-built tools. Given the task and the
current page observation, pick the single best tool from the catalog and extract
its arguments, OR abstain. IMPORTANT: each tool works ONLY on its own `site` --
do NOT pick a tool whose site differs from the task's target website. If no tool
matches BOTH the site AND the intent, abstain. Return ONLY JSON:
{"tool_id": <id or null>, "args": {<name>: <value>}}.

TASK: {task}   OBSERVATION: {obs}
TOOL CATALOG (id, site, use_cond, params, reject_rules): {catalog}
```

G.9 O2: INTENT REJECTION

Checks whether a request violates the supplied rejection rules.

```
You are a safety gate for a reusable web tool. The tool's REJECT RULES list
actions/intents the tool must refuse. Decide whether the USER TASK is asking for
one of these forbidden actions. Judge by INTENT, not shared keywords: e.g.
"look up papers" is NOT the same forbidden action as "look up author profiles".
Return ONLY JSON: {"violates": true|false}.

REJECT RULES: {rules}   USER TASK: {task}
```

G.10 O3: AGENT-STEP EXTRACTION

Requests a specific answer value from the observed page during an agent subgoal.

```
Read the page OBSERVATION and return the specific value the goal asks for, as a
short string (the answer text only, no explanation). If the page does not contain
it, return empty. Return ONLY JSON: {"answer": "<value or empty>"}.

GOAL: {goal}   OBSERVATION: {obs}
```

G.11 E1: INTENT-ANSWER DIAGNOSTIC

Checks whether extracted evidence answers the task for an internal diagnostic, separately from benchmark WebJudge outcomes.

```
Judge whether the EXTRACTED EVIDENCE actually ANSWERS the specific task (not just
lands on a related page). The evidence must contain the concrete result the task
asks for -- e.g. the specific dosage value, the matching listings, the requested
data point -- not merely generic links/articles about the topic. Return ONLY
JSON: {"answers": true|false, "reason": "<short>"}.

TASK: {task}   EXTRACTED EVIDENCE (tool output): {evidence}
```