WebSync: A Self-Evolving Web Agent Framework for Co-Evolving Strategic Advice and Executable Skills

Anonymous ACL submission

Abstract

With the advent of large language models, autonomous web agents have achieved more sophisticated capabilities in planning, reflecting, and tool using, enabling automating navigation in complex and dynamic web environments. Despite this, existing efforts to instill experiential learning through memory-centric retrieval or skill-centric synthesis are respectively limited by computational inefficiency and brittle execution, making it difficult for web agents to reuse experience reliably. To tackle this problem, we introduce a web agent framework called **WebSync**, which is designed to unify strategic advice and skill synthesis through a structured skill recipe to compile experience into executable skills more efficiently and robustly. Furthermore, it enables continuous self-improvement by using runtime feedback to trigger iterative recipe and skill refinement, supporting a transition from trial-and-error reasoning to guarded execution. Experimental results on WebArena and WebVoyager benchmarks demonstrate the effectiveness and versatility of **WebSync** across multiple backbone large language models.

1 Introduction

Large Language Models (LLMs) significantly advance web automation by enabling agents to navigate complex and dynamic environments with sophisticated planning, reflection, and tool-use capabilities (Zhou et al., 2023; He et al., 2024; Deng et al., 2023, 2024; Luo et al., 2025; Ning et al., 2025). Despite these advances, existing web agents lack sufficient experiential learning, trapping them in a "solve-once, forget-again" cycle where routine tasks never become more efficient over time, and agents repeat costly trial-and-error reasoning for familiar scenarios.

To endow web agents with experiential learning, prior research has pursued two distinct paradigms. 1) **Memory-centric approaches** (Ouyang et al., 2025; Wang et al., 2024; Liu et al., 2025a) retrieve prior experiences—execution logs or refined summaries—to guide planning and acting. While improving adaptability, this approach remains computationally expensive as agents must re-interpret unstructured memories and re-reason through each step. 2) **Skill-centric approaches** (Wang et al., 2025; Zheng et al., 2025; Yu et al., 2025) synthesize executable tools directly from successful trajectories. However, skill synthesis remains an isolated trace-to-code pipeline, structurally disconnected from strategic reasoning, yielding tools that lack runtime safeguards and *fail silently* under interface changes.

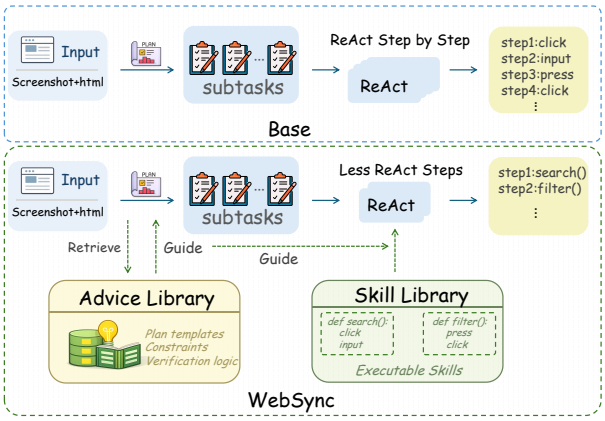

Figure 1: The workflow of WebSync. Unlike the **Base** method (top), which relies on inefficient step-by-step ReAct reasoning for primitive actions (e.g., `click`, `input`), **WebSync** (bottom) significantly reduces interaction steps by integrating strategic advice with executable skills. This allows the agent to transition from granular trial-and-error to robust, high-level function calls (e.g., `search()`).

These two paradigms, though seemingly opposed, capture *complementary* knowledge layers: memory encodes *strategic wisdom*—when to act, what to avoid, and how to recover—providing adaptability, while skills encode *executable procedures* that bypass step-by-step reasoning, provid-

ing efficiency. In isolation, each inherits a critical blind spot: memory without skills forces costly re-derivation of execution plans (*efficiency bottleneck*), while skills without strategic grounding lack awareness of their own applicability boundaries and failure modes (*safety bottleneck*). The gap we target is therefore not skill synthesis alone, but the absence of a persistent mechanism that couples strategic advice, executable bindings, runtime verification, and failure-driven refinement across episodes.

To bridge this gap, we introduce WebSync, a self-evolving framework built around *recipe-mediated co-evolution*. WebSync compiles experiential knowledge into *Skill Recipes*—persistent contracts specifying *when* a skill applies (trigger conditions), *what* it requires (parameter schemas), *how* to verify success (postconditions), and *how* to recover from failures (fallback strategies). These recipes act as the shared interface between an *Advice Library* for strategic knowledge and a *Skill Library* for executable procedures: advice constrains skill generation, recipes govern runtime reuse, and execution feedback refines the same recipe object over time, as shown in Figure 1. This design distinguishes WebSync from prior skill-centric systems that primarily compile successful traces into programs or abstract skill interfaces.

Our contributions are summarized as follows:

- We present **WebSync**, a self-evolving framework that integrates memory-based guidance and executable skills through *recipe-mediated co-evolution*. A Skill Recipe persists as the shared object that links strategic advice, executable bindings, runtime verification, and failure-driven refinement.

- We propose **Recipe-Instructed Skill Synthesis**, a mechanism in which strategic advice constrains skill parameterization and verification. Unlike one-way trace-to-code synthesis, recipe-instructed synthesis uses both successful and failed experience to produce guarded executable procedures with explicit applicability and recovery logic.

- Extensive experiments on **WebArena** and **WebVoyager** across multiple backbone LLMs demonstrate that WEBSYNC consistently outperforms existing methods, achieving up to a 14.5% absolute gain in success rate and a nearly 40% reduction in interaction steps, showing that recipe-guided skills can improve robustness alongside execution efficiency. Our code will be released via `https://anonymous.4open.science/r/WebSync-5DF1/`.

2 Related Work

Web Agents LLM-driven agents have evolved into general-purpose assistants that integrate text and visual signals to navigate complex environments (Yao et al., 2022a; He et al., 2024; Zhou et al., 2023; Deng et al., 2023). To enhance reliability, recent approaches have incorporated supervised fine-tuning (SFT) (Luo et al., 2025; Furuta et al., 2024; Zheng et al., 2024; Lai et al., 2024), reinforcement learning (RL) (Wei et al., 2025; Liu et al., 2025b; Qin et al., 2025; Qi et al., 2024), and explicit planning mechanisms (Zhang et al., 2025b; Fang et al., 2025). However, these systems typically operate in a memory-less manner, treating every task instance as a novel problem. This results in redundant, computationally expensive reasoning for routine activities, underscoring the necessity for experiential learning mechanisms.

Self-Evolving Agents Rather than relying on expensive model re-training, self-evolving agents autonomously refine their capabilities by accumulating experience. By leveraging foundational techniques such as natural language reflection, trajectory optimization, and automated tool synthesis, these systems continuously adapt within open-ended environments (Gao et al., 2025; Zhang et al., 2025a; Zhao et al., 2024; Qiu et al., 2025; Lin et al., 2025). Recent zero-data frameworks such as Agent0 further show that tool-integrated co-evolution can bootstrap agents without human-curated data (**?**); our setting is complementary, focusing on how accumulated web-interaction traces are compacted into reusable recipe contracts for guarded execution. In the domain of autonomous web navigation, such strategies generally manifest in two distinct paradigms: (i) **Memory-Centric Evolution** (Liu et al., 2025a; Ouyang et al., 2025; Wang et al., 2024) which guides planning by retrieving historical insights that range from raw logs to optimized reasoning paths. While adaptable, these methods remain inference-heavy because they require costly re-reasoning even for familiar routines. (ii) **Skill-Centric Evolution** (Zheng et al., 2025; Wang et al., 2025; Yu et al., 2025), which compiles successful trajectories into executable tools.

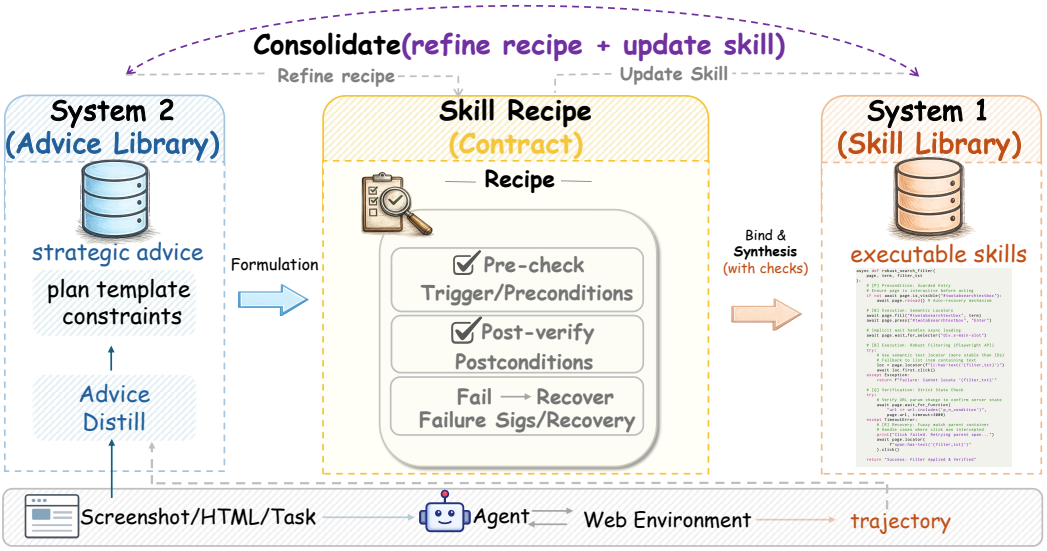

Figure 2: **Overview of WebSync.** It consists of three stages: (1) **Advice Distillation** ($\tau, x \rightarrow A$), where interaction traces are compiled into high-level strategic advice; (2) **Recipe Formulation** ($A \rightarrow \mathcal{S}$), where advice is formalized into a structured contract defining preconditions, postconditions, and recovery logic; and (3) **Recipe-Instructed Skill Consolidation** ($\mathcal{S}, \tau \rightarrow f$), where guarded executable tools are generated. The top loop illustrates the continuous consolidation process that refines recipes and hardens tools based on runtime feedback.

While skill-centric methods demonstrate the value of reusable programs, they largely treat skill synthesis as a one-way process from successful traces to executable code or abstract interfaces. For example, polymorphic skill abstractions such as PolySkill improve transfer by separating abstract skill interfaces from site-specific implementations, but they do not explicitly bind failed traces, runtime verification, and recovery logic into a persistent contract. In contrast, WebSync focuses on *recipe-mediated co-evolution*: a Skill Recipe persists as the shared contract between strategic advice and executable bindings. This contract not only constrains skill synthesis, but also governs runtime applicability, postcondition-based verification, and failure-driven refinement. Thus, the key distinction of WebSync is not simply adding tool specifications, but using recipes as the mechanism that couples memory, execution, and feedback across episodes.

3 WebSync

We use standard web-navigation notation. An interaction episode begins with a natural language instruction x and a starting web state s_0. At each time step t, the agent receives an observation o_t (encompassing rendered content and DOM structure), selects an action a_t, and transitions to a new state s_{t+1}, generating a trajectory $\tau = (o_0, a_0, \ldots, o_T)$. To address the inefficiency of low-level operations, we define a hybrid action space $\mathcal{A} = \mathcal{A}_p \cup \mathcal{A}_f$. Here, $\mathcal{A}_p$ represents primitive browser actions (e.g., click, input), while $\mathcal{A}_f$ consists of parameterized executable skills $f(\theta; o_t)$. Each skill encapsulates a multi-step subroutine, enabling the agent to reason at a strategic level while offloading routine execution.

3.1 Architecture

WebSync reconciles the trade-off between flexible but costly deliberative reasoning and efficient but often brittle executable routines. A core design principle is that once a strategy has been validated through experience, its execution can be offloaded to deterministic procedures equipped with environment-facing verification (e.g., DOM state queries, URL parameter checks), rather than repeatedly re-derived by the model. Rather than treating memory retrieval and tool synthesis as isolated pipelines, our framework unifies them into a continuous loop. Importantly, the entire pipeline—exploration, advice distillation, recipe formulation, and skill synthesis—is executed by *the same backbone LLM* without relying on any stronger external model. The framework operates in an *online* setting: as the agent sequentially processes tasks, it progressively accumulates recipes and skills, following the same continual-learning protocol adopted by prior self-evolving baselines (Zheng et al., 2025; Liu et al., 2025a). As illustrated in Figure 2 and Algorithm 1, this process transforms

Algorithm 1 WebSync Learn–Infer Loop

Require: instruction x, initial state s_0, libraries $\mathcal{M}_A, \mathcal{M}_S$
Ensure: trajectory τ, updated libraries $\mathcal{M}_A, \mathcal{M}_S$

```
τ ← (o_0); E_ref ← ∅
d ← false                                   ▷ episode completion flag
for t = 0, ..., T_max do
    Retrieve recipes S_t ← Retrieve(M_A, x, σ(s_t))
    Plan subgoals g with recipe anchors from S_t
    for each subgoal g_k anchored by recipe S_k do
        B^app_{S_k} ← {f ∈ B_{S_k} : ∀φ ∈ P_{S_k}, φ(o_t) = true}
        if B^app_{S_k} = ∅ then
            a_t ← ReAct(o_t, g_k)
        else
            Ground θ using Θ_{S_k}; a_t ← f(θ; o_t)
        end if
        o_{t+1} ← ExecuteOnEnv(a_t); τ ← τ ⊕ (a_t, o_{t+1})
        if a_t is a skill invocation and Q_{S_k}(o_{t+1}) fails then
            ψ ← MatchFailure(F_{S_k}, o_{t+1})
            If available, execute recovery R_{S_k}[ψ], update o_{t+1}, and append it to τ
            Add (S_k, ψ, o_t, o_{t+1}) to refinement queue E_ref
        end if
        if task is complete then
            d ← true; break
        end if
    end for
    if d = true then
        break
    end if
end for
Distill advice A from trajectory τ and update M_A
Select recurring/useful/stable candidate traces from τ
Formulate or refine recipe S using A and E_ref
Synthesize skill f from S and commit it to M_S only if validation passes
return τ, M_A, M_S
```

ephemeral interaction traces into reusable capabilities through a closed loop of recipe-guided inference and post-episode consolidation.

Advice Distillation. Initially, the agent explores the environment through deliberative reasoning, generating an interaction trajectory $\tau = (o_0, a_0, \ldots, o_T)$ given an instruction x. Crucially, WebSync learns not only from success but also from failure. The framework distills the experience into the **Advice Library** ($\mathcal{M}_A$), where successful traces (τ^+) provide reusable procedural templates ("how to succeed"), while failed traces (τ^-) contribute critical constraints and failure patterns ("what to avoid"), such as hidden overlays, timing issues, or unclickable elements:

$$A \leftarrow \text{Distill}(\tau, x), \quad \mathcal{M}_A \leftarrow \mathcal{M}_A \cup \{A\} \tag{1}$$

This dual-source mechanism converts unstructured interaction logs into structured procedural knowledge that captures both effective strategies and their boundary conditions.

Skill Recipe Formulation. To operationalize the abstract advice A, the system formalizes it into a Skill Recipe (S). Unlike a simple prompt or a static tool interface, S is a persistent contract containing a canonical name (n) and seven operational fields: applicability triggers (σ), preconditions ($\mathcal{P}$), parameter schema (Θ), postconditions ($\mathcal{Q}$), failure signatures ($\mathcal{F}$), recovery logic ($\mathcal{R}$), and executable bindings ($\mathcal{B}$). For example, $\mathcal{P}$ may require a search box to be visible, while $\mathcal{Q}$ may verify a cart-count or URL-state change; the full schema is provided in Appendix A, Table 5. We use Θ for the argument schema and θ for concrete grounded arguments at runtime.

$$S = \langle n, \sigma, \mathcal{P}, \Theta, \mathcal{Q}, \mathcal{F}, \mathcal{R}, \mathcal{B} \rangle \leftarrow \text{Formulate}(A) \tag{2}$$

This tuple serves as the governing specification: $\mathcal{P}$ and $\mathcal{Q}$ define valid execution boundaries, $\mathcal{F}$ records detectable failure modes, $\mathcal{R}$ provides recovery logic, and $\mathcal{B}$ stores validated executable implementations. The same recipe object constrains synthesis, governs runtime checks, and is later refined from failures, thereby coupling strategic memory with executable skills across episodes.

Skill Synthesis. Finally, WEBSYNC synthesizes an executable skill function f for the Skill Library ($\mathcal{M}_S$). Rather than naively cloning a demonstrated trajectory, the synthesizer uses S as a guiding specification to reconstruct a guarded procedure from the raw trace τ, embedding $\mathcal{P}$ as runtime assertions and $\mathcal{R}$ as exception handlers:

$$\begin{aligned} &f(\cdot; \Theta) \leftarrow \text{Synthesize}(\tau \mid S) \\ &\text{commit } f \text{ only if } \text{Validate}(f, \mathcal{Q}) = \text{true} \end{aligned} \tag{3}$$

The resulting tool is committed to $\mathcal{M}_S$ only after passing validation against $\mathcal{Q}$, encouraging reuse through deterministic execution when the recipe conditions are satisfied. We detail the full consolidation pipeline in §3.2.

3.2 Recipe-Instructed Skill Consolidation

Consolidation converts candidate traces into reusable skills while using the Skill Recipe as the contract that guards synthesis, validation, and later refinement. The process has four stages:

Candidate Trace Selection. The agent logs successful subgoal segments $\tau = (o_{t_0}, a_{t_0}, \ldots, o_{t_1})$ as synthesis candidates, but promotes only segments

that satisfy three reuse heuristics: *recurrence* of the intent/signature pair, *utility* in expected step savings, and *stability* of inferable argument slots.

Evidence-Based Recipe Refinement. Before code synthesis, accumulated failure evidence refines the recipe so that observed failures tighten the contract rather than being discarded:

- **Tightening Preconditions ($\mathcal{P}$):** If prior failures were caused by missing affordances (e.g., an absent filter panel), predicates are added to explicitly screen out such invalid contexts.
- **Strengthening Postconditions ($\mathcal{Q}$):** To prevent false positives, verification logic is updated to require stricter evidence, such as specific text updates or DOM state changes.

Constraint-Injecting Skill Synthesis. The synthesizer uses the refined recipe to define the function signature from Θ, inject $\mathcal{P}$ and $\mathcal{Q}$ as runtime assertions, and translate failure signatures $\mathcal{F}$ into recovery branches $\mathcal{R}$.

Recipe-Based Validation. New tools are committed to the Skill Library ($\mathcal{M}_S$) only if they satisfy the recipe's postconditions $\mathcal{Q}$ in matching contexts, reducing the chance that defective skills pollute memory.

3.3 Recipe-Informed Inference

At inference time, recipes couple retrieval, planning, and execution into a *capability-aligned* pipeline.

Retrieval and Planning. At each step t, given instruction x and state signature $\sigma(s_t)$, the agent retrieves relevant recipes $\mathcal{S}_t \leftarrow \text{Retrieve}(\mathcal{M}_A, x, \sigma(s_t))$, which provide plan templates and executable predicates for applicability. The agent then generates a short-horizon plan $\mathbf{g} = (g_1, \ldots, g_K)$ with candidate skill anchors for each subgoal:

$$\mathbf{g}, \{S_k\} \leftarrow \text{Plan}(x, o_t, \mathcal{S}_t). \tag{4}$$

Tool Mapping and Grounding. For each subgoal, the agent filters candidate tools by precondition checks:

$$\mathcal{B}_S^{\text{app}} = \{f \in \mathcal{B}_S \mid \forall \phi \in \mathcal{P}_S,\ \phi(o_t) = \text{true}\}. \tag{5}$$

If $\mathcal{B}_S^{\text{app}}$ is empty, the agent falls back to primitives, preventing execution in invalid states. Arguments θ are grounded according to schema Θ_S and validated against type constraints before invocation.

Benchmark	# Tasks	Domains
WEBARENA	655	4 self-built websites
WEBVOYAGER	643	15 live sites

Table 1: Statistics of the benchmarks. We use the standard subset of 655 tasks for WebArena and the full set of 643 tasks for WebVoyager.

Execution with Verification. During execution, WEBSYNC monitors progress via *environment-facing checks*—DOM queries, URL pattern matching, and element attribute verification—rather than LLM-based textual judgments. This enables deterministic detection of many observable execution failures that text-based reasoning could miss: It evaluates $\mathcal{Q}_S$ on the resulting observation $o_{t'}$ to verify success (e.g., `aria-checked` attributes or cart-count changes). If verification fails, it matches a failure signature $\psi \in \mathcal{F}_S$ and applies the corresponding recovery action $\mathcal{R}_S[\psi]$ when available.

4 Experiments

4.1 Setup

Evaluation Benchmarks. We evaluate WebSync on two complementary benchmarks. First, we use WEBARENA (Zhou et al., 2023), a self-hosted environment, where we strictly follow the standard protocol by focusing on the 655 tasks across four domains (Shopping, Admin, GitLab, Reddit) to isolate reasoning capabilities. Second, to test robustness in open-world conditions, we employ WEBVOYAGER (He et al., 2024) on the live web, handling dynamic content via the `browser-use` interface. The details of dataset statistics are presented in Table 1.

Base Models & Baselines. Our web agent operates within a standard ReAct-style (Yao et al., 2022b) loop. For controlled comparison on WEBARENA, we use CLAUDE-3.7-SONNET (Anthropic, 2024) as the primary backbone and additionally evaluate on GPT-4.1 and GLM-4.5 to verify cross-model generalization. On WEBVOYAGER, we evaluate with GPT-4O (Hurst et al., 2024), CLAUDE-3.7-SONNET, QWEN3-VL-8B (Bai et al., 2025), and QWEN2.5-VL-7B. We compare against representative self-evolving web agent frameworks spanning both paradigms: memory-centric methods AWM (Wang et al., 2024) and REASONINGBANK (Ouyang et al., 2025), and

Method	Shopping		Admin		Gitlab		Reddit		Overall	
	SR	*Step*	*SR*	*Step*	*SR*	*Step*	*SR*	*Step*	*SR*	*Step*
				Claude-3.7-Sonnet						
Claude-3.7-Sonnet (Anthropic, 2024)	38.5	6.1	49.5	8.4	36.7	10.6	53.8	5.5	43.5	7.9
AWM (Wang et al., 2024)	39.6	7.2	47.8	9.3	34.6	10.9	52.8	7.0	42.6	8.8
SkillWeaver (Zheng et al., 2025)	46.2	4.4	52.7	5.3	55.1	7.2	72.5	5.0	54.7	5.5
ASI (Wang et al., 2025)	59.1	3.6	61.3	5.6	64.2	6.4	78.5	4.8	64.3	5.1
PolySkill (Yu et al., 2025)	58.3	3.3	63.5	–	62.5	–	80.4	–	64.5	–
ReasoningBank (Ouyang et al., 2025)	44.9	5.6	53.3	7.6	41.1	9.5	57.5	5.2	48.2	7.2
WebSync (Ours)	**64.2**	**3.1**	**67.5**	**4.2**	**65.8**	**5.5**	**81.8**	**4.7**	**68.4**	**4.3**
				GPT-4.1						
GPT-4.1 (Hurst et al., 2024)	37.4	8.3	44.0	8.8	38.9	11.0	66.0	6.0	44.3	8.3
SkillWeaver (Zheng et al., 2025)	39.3	4.8	48.2	5.8	50.3	8.0	71.2	5.8	50.0	6.1
ASI (Wang et al., 2025)	46.3	4.0	53.6	6.0	46.8	7.0	73.7	5.5	52.9	5.6
PolySkill (Yu et al., 2025)	51.4	–	54.8	–	54.2	–	73.2	–	56.6	–
WebSync (Ours)	**54.5**	**3.4**	**58.2**	**4.5**	**57.5**	**5.8**	**76.4**	**4.8**	**59.9**	**4.6**
				GLM-4.5						
GLM-4.5 (**?**)	36.1	8.7	40.9	9.2	37.5	11.5	59.4	6.5	41.6	8.7
SkillWeaver (Zheng et al., 2025)	34.2	5.2	39.7	6.2	39.1	8.5	54.8	6.4	40.4	6.6
ASI (Wang et al., 2025)	38.6	4.5	43.7	6.5	40.8	7.5	62.5	6.0	44.5	6.1
PolySkill (Yu et al., 2025)	39.4	–	44.5	–	41.6	–	63.2	–	45.3	–
WebSync (Ours)	**42.5**	**3.8**	**47.8**	**5.0**	**45.5**	**6.2**	**66.8**	**5.2**	**48.7**	**5.0**

Table 2: **Results on WebArena** (655 tasks across 4 domains). SR (↑) denotes success rate and Step (↓) denotes average interaction steps. Overall SR and Step are the weighted averages over the four domains based on the number of tasks. "–" indicates results not reported or not available (e.g., closed-source method).

skill-centric methods SKILLWEAVER (Zheng et al., 2025), POLYSKILL (Yu et al., 2025), and ASI (Wang et al., 2025). On WebVoyager, we additionally compare with WEBCOACH (Liu et al., 2025a). For WebArena, most baseline results are adopted directly from the respective original papers under aligned evaluation protocols; however, for SkillWeaver and ASI, we reran the methods ourselves in order to obtain their step-level trajectories. For WebVoyager, all results (both baselines and WebSync) are from our own runs using the Browser Use Agent framework with identical settings. Tasks are evaluated sequentially in the default index order of the official task files, with no curriculum or custom ordering applied.

Metrics Across both benchmarks, we focus on three primary metrics: (1) **Success Rate (SR)**, measuring effective task completion; (2) **Average Interaction Steps**, quantifying efficiency; and (3) **Skill Utilization Rate (SUR)**, defined as the proportion of tool invocations among all executed actions. SUR serves as a diagnostic proxy for how often the agent reuses synthesized skills instead of primitive actions.

4.2 Main Results

Results on WebArena. As presented in Table 2, WEBSYNC achieves the strongest WebArena result in our comparison with CLAUDE-3.7-SONNET, reaching 68.4% overall SR. Compared to **memory-centric baselines** (AWM, REASONING BANK), which still require costly on-the-fly interpretation and average 7.2–8.8 steps, WEBSYNC compiles repeated routines into executable skills and reduces interaction cost to 4.3 steps, cutting trajectory length by nearly half while elevating SR by over 20 points. Against **skill-centric baselines** (POLYSKILL, ASI), which leverage tool synthesis but lack recipe-level runtime contracts, WEBSYNC surpasses the strongest prior method by ∼4% SR through guarded skill recipes. This improvement does not come at the cost of more interactions: in Shopping—the only subset where POLYSKILL reports steps—WEBSYNC achieves both higher SR (64.2% vs. 58.3%) and fewer steps (3.1 vs. 3.3).

Cross-Backbone Generalization on WebArena. We further evaluate with GPT-4.1 and GLM-4.5 (Table 2). WEBSYNC consistently achieves the highest SR, outperforming the strongest baseline by 3.3% and 3.4% respectively, suggesting that the gains stem from recipe-guided skills rather than

Method	SR (%) ↑	Steps ↓
(a) Baseline Comparison (GPT-4o)		
GPT-4o	65.3	10.9
SkillWeaver (Zheng et al., 2025)	68.5	8.4
WebCoach (Liu et al., 2025a)	69.8	9.2
WebSync (Ours)	**72.8**	**6.5**

	SR (%) ↑		Steps ↓	
Backbone	Base	WebSync	Base	WebSync
(b) Cross-Backbone Generalization				
GPT-4o	65.3	**72.8**	10.9	**6.5**
Claude-3.7-Sonnet	76.0	**86.3**	9.8	**5.4**
Qwen3-VL-8B	42.1	**56.6**	16.4	**11.9**
Qwen2.5-VL-7B	32.8	**35.2**	17.4	**14.2**

Table 3: **Results on WebVoyager** (15-site average). **(a)** Baseline comparison under GPT-4o against self-evolving methods; **(b)** Cross-backbone generalization: WebSync consistently improves SR and reduces steps across all four backbones. Per-site breakdowns are in Appendix C.

backbone-specific properties.

Results on WebVoyager. Table 3 summarizes results on WebVoyager, where agents face dynamic content and unpredictable UI interruptions. Under GPT-4o, WebSync achieves 72.8% SR with only 6.5 steps, outperforming WebCoach by 3.0% SR while requiring 29% fewer steps. SkillWeaver reduces steps (8.4) but lags in SR (68.5%), suggesting that step reduction without runtime verification is insufficient. The results also demonstrate strong cross-model generalization (Table 3b). On Claude-3.7-Sonnet, WebSync boosts SR from 76.0% to 86.3% (+10.3%). The most pronounced gain appears on Qwen3-VL-8B (+14.5%, from 42.1% to 56.6%), suggesting that recipe-encoded procedural knowledge effectively compensates for smaller models' reasoning limitations. Per-site breakdowns are in Appendix C.

Cost and Amortized Efficiency. Appendix F further reports a token-level cost decomposition. While WebSync pays additional skill-consolidation and recipe-refinement overhead during learning, it reduces recurring action-generation tokens from 55,367.4 to 48,184.6 per task on average, indicating that the learned recipes primarily improve amortized efficiency over repeated web operations.

4.3 Ablation Study

To assess the distinct contributions of each component within WebSync, we conducted an ablation study across both benchmarks. We compare the full framework against component-level variants, analyzing the impact on Success Rate (SR), efficiency (Average Steps), and Skill Utilization Rate (SUR). We additionally include two targeted WebArena ablations to isolate contracts without persistent memory and the effect of runtime refinement. The results are summarized in Table 4.

Impact of Executable Skills. The variant *w/o Executable Skills* removes the Skill Library ($\mathcal{M}_S$), forcing the agent to rely solely on primitive actions via ReAct reasoning. On WebArena, this leads to a marked decline in performance, with SR dropping from 68.4% to 55.2% and average steps increasing from 4.3 to 6.8. The same trend appears on WebVoyager (72.8% to 65.2% SR; 6.5 to 9.8 steps), indicating that encapsulating repeated subroutines into executable skills is important for reducing long-horizon error accumulation.

Impact of Strategic Advice. The *w/o Strategic Advice* variant bypasses the advice distillation phase, synthesizing tools directly from interaction traces without high-level semantic constraints. A key observation is the sharp drop in SUR on WebArena, which falls from 28.9% to 14.2%. This indicates that without the generalization provided by strategic advice, synthesized tools tend to overfit to specific trajectory instances (e.g., binding to transient element IDs). Consequently, these brittle tools are rarely triggered in future episodes, forcing the agent to revert to inefficient primitive actions and lowering WebArena SR from 68.4% to 60.8%.

Impact of Skill Recipes. The *w/o Skill Recipes* variant retains the libraries but disables the P-B-Q-R specification checks during inference. On WebArena, this variant maintains a relatively high SUR (24.1%), yet SR decreases from 68.4% to 65.3%. This discrepancy highlights the danger of "blind execution." Without the guardrails of Preconditions ($\mathcal{P}$) and Postconditions ($\mathcal{Q}$), the agent frequently invokes tools in invalid states or fails to detect observable execution errors. While the agent still uses skills frequently, the lack of runtime verification compromises the actual outcome, showing that specification quality is essential for turning skill usage into reliable task completion.

Method	WebArena			WebVoyager		
	SR (%)	Steps	SUR (%)	SR (%)	Steps	SUR (%)
WebSync	**68.4**	**4.3**	**28.9**	**72.8**	**6.5**	**31.5**
Inline P/Q/R (no lib.)	49.8	7.2	–	–	–	–
w/o Executable Skills	55.2	6.8	–	65.2	9.8	–
w/o Strategic Advice	60.8	5.2	14.2	66.8	8.7	15.8
w/o Skill Recipes	65.3	4.8	24.1	69.5	7.8	26.4
w/o Runtime Refinement	66.0	4.6	28.5	–	–	–

Table 4: **Ablation Analysis across Benchmarks.** We compare the full WebSync framework against component-level variants on WebArena and WebVoyager. The two targeted variants, *Inline P/Q/R (no lib.)* and *w/o Runtime Refinement*, are evaluated on WebArena with the Claude-3.7-Sonnet backbone. We introduce **Skill Utilization Rate (SUR)** to measure the proportion of steps executed via synthesized tools. The results highlight that strategic advice significantly boosts tool reusability, while specification constraints ensure their robust execution.

Persistent Recipes vs. Inline Contracts. The *Inline P/Q/R, no library* variant asks the same backbone to draft preconditions, postconditions, and recovery logic from the current instruction and state, but does not store them in either the Advice Library or the Skill Library. This improves the WebArena base ReAct agent from 43.5% to 49.8% SR, showing that contract prompting alone is useful. However, full WebSync remains much stronger (68.4% SR and 4.3 steps), indicating that the main gain comes from persistent recipes that are distilled, validated, reused, and refined across tasks.

Impact of Runtime Refinement. The *w/o Runtime Refinement* variant freezes recipes after initial validation and disables evidence-based updates from later failures. On WebArena, this reduces SR from 68.4% to 66.0% and increases steps from 4.3 to 4.6 while keeping SUR similar (28.5% vs. 28.9%). This isolates the value of failure-driven refinement: runtime refinement does not merely increase skill usage, but improves when and how existing skills remain reliable under interface variation.

Self-Evolution vs. External Synthesis. We additionally investigate whether delegating skill synthesis to a stronger external model improves over self-evolution. Results (Appendix D) show that the gain is marginal (+1.4% SR), suggesting that the self-evolving pipeline captures most of the benefit without requiring an external model.

4.4 Case Study

We illustrate WebSync on Amazon product filtering, a task that is brittle due to asynchronous UI loading and nested DOM structures. The retrieved advice identifies Amazon checkboxes as prone to unclickable overlays, so the synthesized skill follows a P-B-Q-R contract: it first checks that the refinement panel is visible, executes the filter action with semantic locators, verifies success through observable page state such as `aria-checked` or active filter tags, and triggers a fuzzy parent-container click when exact matching fails. Unlike a replay-style skill, this guarded procedure does not assume that a click succeeded; it returns control only after postcondition verification or reports failure after recovery is exhausted. This example highlights the role of recipes as the interface between strategy and execution: advice supplies the failure pattern, while the executable binding enforces it through concrete checks. It also shows why recovery is kept inside the skill rather than delegated to another high-level planning loop, reducing repeated trial-and-error after transient UI failures. This allows WebSync to handle layout shifts internally and avoid an expensive high-level replanning cycle. The full pseudocode (Algorithm A.4) and trajectory comparison are provided in Appendix H.

5 Conclusion

We introduced WEBSYNC, a self-evolving framework that transforms transient reasoning into reusable execution skills. Unlike prior approaches that isolate memory from tool synthesis, WEBSYNC unifies them through *Skill Recipes*—formal contracts that encode applicability and verification logic. This mechanism allows agents to synthesize guarded tools, bridging flexible planning and reliable execution. Empirical results on WebArena and WebVoyager show consistent gains over representative baselines across multiple backbone models.

Limitations

Expressivity of Text-Based recipes. Our current Skill recipes primarily rely on text-based or DOM-selector predicates, such as `visible(selector)`. This symbolic abstraction, however, is ill-equipped to capture tasks that are heavily dependent on visual semantics or spatial dynamics, for instance, instructions to "drag the slider" or "select the visually highlighted item." Even though the multimodal backbone is capable of perceiving these elements, the intermediate recipe's reliance on text limits the system's capacity to formally codify and verify purely visual constraints.

Cold Start and Exploration Costs. Since WebSync's self-improvement is predicated on the accumulation of useful trajectories, the system cannot synthesize effective skills for novel domains or long-tail tasks where reusable evidence is scarce. In these scenarios, the agent must rely more heavily on deliberative trial-and-error reasoning. Mitigating this "cold start" overhead through cross-domain transfer or few-shot recipe seeding remains an important direction for future research.

Real-World Anti-Bot Constraints. As evidenced by our WebVoyager results, particularly on Google Flights, the agent encounters difficulties with strict anti-bot measures like CAPTCHAs and complex authentication flows. While this challenge is pervasive in web agent research, the high-frequency execution pattern characteristic of synthesized skills may arguably increase the probability of triggering such defenses compared to slower, more human-like browsing behaviors.

References

Anthropic. 2024. Developing a computer use model. https://www.anthropic.com/news/developing-computer-use. Accessed: 2025-04-12.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, and 45 others. 2025. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*.

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

Yang Deng, An Zhang, Yankai Lin, Xu Chen, Ji-Rong Wen, and Tat-Seng Chua. 2024. Large language model powered agents in the web. In *Companion Proceedings of the ACM Web Conference 2024*, pages 1242–1245.

Tianqing Fang, Hongming Zhang, Zhisong Zhang, Kaixin Ma, Wenhao Yu, Haitao Mi, and Dong Yu. 2025. WebEvolver: Enhancing web agent self-improvement with co-evolving world model. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 8970–8986, Suzhou, China. Association for Computational Linguistics.

Hiroki Furuta, Kuang-Huei Lee, Ofir Nachum, Yutaka Matsuo, Aleksandra Faust, Shixiang Gu, and Izzeddin Gur. 2024. Multimodal web navigation with instruction-finetuned foundation models. In *International Conference on Representation Learning*, volume 2024, pages 29528–29558.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, and 1 others. 2025. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*.

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

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Hanyu Lai, Xiao Liu, Iat Long Iong, Shuntian Yao, Yuxuan Chen, Pengbo Shen, Hao Yu, Hanchen Zhang, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. 2024. Autowebglm: A large language model-based web navigating agent. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*, pages 5295–5306. ACM.

Jiaye Lin, Yifu Guo, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, Mingguang Chen, Hongzhang Liu, Ronghao Chen, Yangfan He, and 1 others. 2025. Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents. *arXiv preprint arXiv:2508.02085*.

Genglin Liu, Shijie Geng, Sha Li, Hejie Cui, Sarah Zhang, Xin Liu, and Tianyi Liu. 2025a. Webcoach: Self-evolving web agents with cross-session memory guidance. *arXiv preprint arXiv:2511.12997*.

Jiarun Liu, Jia Hao, Chunhong Zhang, and Zheng Hu. 2025b. Wepo: Web element preference optimization for llm-based web navigation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 26614–26622.

Haohao Luo, Jiayi Kuang, Wei Liu, Ying Shen, Jian Luan, and Yang Deng. 2025. Browsing like human: A multimodal web agent with experiential fast-and-slow thinking. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14232–14251.

Liangbo Ning, Ziran Liang, Zhuohang Jiang, Haohao Qu, Yujuan Ding, Wenqi Fan, Xiao-yong Wei, Shanru Lin, Hui Liu, Philip S Yu, and 1 others. 2025. A survey of webagents: Towards next-generation ai agents for web automation with large foundation models. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 6140–6150.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, and 1 others. 2025. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Wenyi Zhao, Yu Yang, Xinyue Yang, Jiadai Sun, Shuntian Yao, and 1 others. 2024. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. *arXiv preprint arXiv:2411.02337*.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, and 1 others. 2025. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, and 1 others. 2025. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*.

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

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

Zhepei Wei, Wenlin Yao, Yao Liu, Weizhi Zhang, Qin Lu, Liang Qiu, Changlong Yu, Puyang Xu, Chao Zhang, Bing Yin, and 1 others. 2025. Webagent-r1: Training web agents via end-to-end multi-turn reinforcement learning. *arXiv preprint arXiv:2505.16421*.

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

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

Simon Yu, Gang Li, Weiyan Shi, and Peng Qi. 2025. Polyskill: Learning generalizable skills through polymorphic abstraction. *arXiv preprint arXiv:2510.15863*.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, and 1 others. 2025a. Agent learning via early experience. *arXiv preprint arXiv:2510.08558*.

Yao Zhang, Zijian Ma, Yunpu Ma, Zhen Han, Yu Wu, and Volker Tresp. 2025b. Webpilot: A versatile and autonomous multi-agent system for web task execution with strategic exploration. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 23378–23386.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19632–19642.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and 1 others. 2025. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*.

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

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

A Recipe Schema and Procedure Pseudocode

Table 5 summarizes the field-level schema of a Skill Recipe.

Field	Description & Logic
Name (n)	Canonical skill name (e.g., `search`, `apply_filter`, `add_to_cart`).
Trigger (σ)	Domain + URL patterns + page markers for applicability indexing.
Preconditions ($\mathcal{P}$)	Predicates that must hold before execution (e.g., `is_visible(search_box)`).
Params (Θ)	Argument schema with types, constraints, and grounding hints.
Postconditions ($\mathcal{Q}$)	Verifiable success evidence (e.g., `count_changed(cart)`).
Failure Sigs ($\mathcal{F}$)	Detectable failure predicates (modal, login redirect, timeout).
Recovery ($\mathcal{R}$)	Failure $\rightarrow$ recovery actions (close modal, wait/retry).
Bindings ($\mathcal{B}$)	Tool implementations with versions and performance statistics.

Table 5: **Skill Recipe Card.** The skill recipe S standardizes the intent, applicability, and verification logic, serving as the governing contract for the implementation.

This section provides procedure-level pseudocode for the three core stages of WebSync. Algorithm A.1 describes advice distillation from successful and failed trajectories. Algorithm A.2 details recipe-instructed skill consolidation. Algorithm A.3 presents recipe-guided inference with runtime refinement signals.

Algorithm A.1 Advice Distillation

Require: trajectory $\tau = (o_0, a_0, \ldots, o_T)$, instruction x, Advice Library $\mathcal{M}_A$
Ensure: updated Advice Library $\mathcal{M}_A$
$y \leftarrow \text{ClassifyOutcome}(\tau)$ ▷ success or failure
if $y = \text{success}$ **then**
  $A_{raw} \leftarrow \text{ExtractTemplate}(\tau, x)$ ▷ how to succeed
else
  $A_{raw} \leftarrow \text{ExtractFailurePattern}(\tau, x)$ ▷ what to avoid
end if
$A \leftarrow \text{Distill}(A_{raw}, \tau, x)$
$\mathcal{M}_A \leftarrow \mathcal{M}_A \cup \{A\}$
return $\mathcal{M}_A$

Algorithm A.2 Recipe-Instructed Skill Consolidation

Require: candidate trace τ_{cand}, instruction x, Advice Library $\mathcal{M}_A$, Skill Library $\mathcal{M}_S$, refinement evidence $\mathcal{E}_{ref}$
Ensure: updated Skill Library $\mathcal{M}_S$
if not $(\text{Recurrence}(\tau_{cand}) \wedge \text{Utility}(\tau_{cand}) \wedge \text{Stability}(\tau_{cand}))$ **then**
  return $\mathcal{M}_S$ ▷ filter one-off or unstable traces
end if
$A \leftarrow \text{Retrieve}(\mathcal{M}_A, x, \sigma(\tau_{cand}))$
$S = \langle n, \sigma, \mathcal{P}, \Theta, \mathcal{Q}, \mathcal{F}, \mathcal{R}, \mathcal{B} \rangle \leftarrow \text{Formulate}(A)$
for each prior failure $\psi \in \text{FailureRecords}(\tau_{cand}, \mathcal{E}_{ref})$ **do**
  if ψ indicates a missing affordance **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{\text{predicate}(\psi)\}$
  end if
  if ψ indicates a false-positive verification **then**
    $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\text{stricter_predicate}(\psi)\}$
  end if
  $\mathcal{F} \leftarrow \mathcal{F} \cup \{\text{signature}(\psi)\}$
  $\mathcal{R}[\psi] \leftarrow \text{RecoveryAction}(\psi)$
end for
$f(\cdot; \Theta) \leftarrow \text{Synthesize}(\tau_{cand} \mid S)$
if $\text{Validate}(f, \mathcal{Q}) = \text{true}$ **then**
  $\mathcal{B} \leftarrow \mathcal{B} \cup \{\langle f, \text{UtilityStat}(f) \rangle\}$
  $\mathcal{M}_S \leftarrow \text{Upsert}(\mathcal{M}_S, \langle S, \mathcal{B} \rangle)$
else
  discard f
end if
return $\mathcal{M}_S$

B Prompts

To ensure reproducibility, we provide the detailed system prompts used in WebSync. Please refer to the Figure 3, Figure 5 and Figure 4 for details.

C Detailed WebVoyager Results

Table 6 provides the per-site success rate breakdown on WebVoyager for three backbones (GPT-4o, Claude-3.7-Sonnet, Qwen3-VL-8B). WebSync consistently improves over the base agent across all 15 websites.

D Self-Evolution vs. External Synthesis

To investigate whether a stronger external model can further boost skill quality, we compare self-evolution (the full pipeline run by QWEN3-VL-8B alone) against a variant where Advice Distillation and Skill Synthesis are delegated to GPT-4O while the 8B model only executes the resulting recipes at inference.

As shown in Table 7, the external-synthesizer variant yields 58.0% SR and 11.2 steps, compared to 56.6% SR and 11.9 steps for self-evolution. The marginal gain (+1.4% SR) confirms that while larger models produce slightly higher-quality recipes, the self-evolving pipeline is already highly effective—the 8B-only setup achieves the vast majority of the benefit (14.5% absolute gain over the 42.1% baseline) without requiring any external model.

	GPT-4o		Claude-3.7-Sonnet		Qwen3-VL-8B	
Task Website	Base	**WebSync** (Δ)	Base	**WebSync** (Δ)	Base	**WebSync** (Δ)
Allrecipes	88.9	**93.3** (+4.4)	93.3	**100.0** (+6.7)	46.7	**62.2** (+15.5)
Amazon	95.1	**100.0** (+4.9)	97.6	**100.0** (+2.4)	65.9	**78.0** (+12.1)
Apple	60.5	**74.4** (+13.9)	72.1	**83.7** (+11.6)	32.6	**48.8** (+16.2)
ArXiv	83.7	**90.7** (+7.0)	88.4	**95.3** (+6.9)	34.9	**48.8** (+13.9)
BBC News	71.4	**83.3** (+11.9)	78.6	**88.1** (+9.5)	31.0	**47.6** (+16.6)
Booking	13.6	**25.0** (+11.4)	25.0	**38.6** (+13.6)	18.2	**31.8** (+13.6)
Cambridge Dict.	95.3	**97.7** (+2.4)	97.7	**100.0** (+2.3)	74.4	**88.4** (+14.0)
Coursera	59.5	**69.0** (+9.5)	76.2	**85.7** (+9.5)	57.1	**69.0** (+11.9)
ESPN	65.9	**79.5** (+13.6)	75.0	**88.6** (+13.6)	29.5	**45.5** (+16.0)
GitHub	85.4	**92.7** (+7.3)	90.2	**95.1** (+4.9)	43.9	**61.0** (+17.1)
Google Flights	7.1	**19.0** (+11.9)	19.0	**33.3** (+14.3)	4.8	**16.7** (+11.9)
Google Map	87.8	**92.7** (+4.9)	92.7	**97.6** (+4.9)	58.5	**73.2** (+14.7)
Google Search	23.3	**37.2** (+13.9)	34.9	**48.8** (+13.9)	39.5	**53.5** (+14.0)
Huggingface	67.4	**76.7** (+9.3)	74.4	**86.0** (+11.6)	39.5	**55.8** (+16.3)
Wolfram Alpha	76.1	**84.8** (+8.7)	84.8	**93.5** (+8.7)	54.3	**69.6** (+15.3)
Avg. SR	65.3	**72.8 (+7.5)**	76.0	**86.3 (+10.3)**	42.1	**56.6 (+14.5)**
Avg. Steps	10.9	**6.5 (-4.4)**	9.8	**5.4 (-4.4)**	16.4	**11.9 (-4.5)**

Table 6: **Per-Site Success Rate Breakdown on WebVoyager.** Columns for WebSync show absolute performance followed by improvement over Base (Δ).

Setting	SR (%) ↑	Steps ↓
Base	42.1	16.4
WebSync (self only)	56.6	11.9
WebSync + GPT-4o synth.	**58.0**	**11.2**

Table 7: Self-evolution vs. external synthesis on WebVoyager (Qwen3-VL-8B, 15-site average).

Category	Setting	Value
Browser	Window / viewport size	1920 × 1080
	Headless mode	true
	Min. wait (page load)	0.5 s
	Wait between actions	0.5 s
	Wait for network idle	0.5 s
Agent	Max steps per episode	30
	Max actions per step	10
	Max consecutive failures	3
	Use vision	true
	Vision detail level	auto
	Use thinking	true

Table 8: Hyperparameter configuration for WebVoyager experiments. All methods (baselines and WebSync) use identical settings.

E Experimental Setup Details

For full reproducibility, we report the hyperparameters and environmental configurations used in our experiments.

WebArena. We strictly follow the official evaluation scripts and default configurations provided by the WebArena benchmark. No custom hyperparameters are introduced; our setup is fully aligned with how other baselines evaluate on WebArena. We use the standard subset of 655 tasks across four domains (Shopping, Admin, GitLab, Reddit). Tasks are evaluated sequentially in the default index order of the official task files, with no curriculum or custom ordering applied.

WebVoyager. All experiments on WebVoyager, including baselines (SkillWeaver and WebCoach) and WebSync, are conducted using the Browser Use Agent framework under identical settings. Table 8 lists the complete configuration. Tasks are evaluated sequentially in the default index order of the official task JSON files.

Baseline Sources. For WebArena, most baseline results are adopted directly from the respective original papers under aligned evaluation protocols; however, for SkillWeaver and ASI, we reran the methods ourselves in order to obtain their step-level trajectories. For WebVoyager, all results (both baselines and WebSync) are from our own runs using the Browser Use Agent framework with identical settings. Tasks are evaluated sequentially in the default index order of the official task files, with no curriculum or custom ordering applied.

F Cost Analysis

To evaluate the economic feasibility of WebSync, we conduct a token-level cost analysis in addition to the step-based efficiency results reported

Algorithm A.3 Recipe-Guided Inference with Runtime Refinement

Require: instruction x, initial state s_0, Advice Library $\mathcal{M}_A$, Skill Library $\mathcal{M}_S$
Ensure: trajectory τ, refinement evidence $\mathcal{E}_{ref}$

$\tau \leftarrow (o_0)$; $\mathcal{E}_{ref} \leftarrow \emptyset$
for $t = 0, \ldots, T_{\max}$ **do**
  $\mathcal{S}_t \leftarrow \mathrm{Retrieve}(\mathcal{M}_A, x, \sigma(s_t))$
  $\mathbf{g}, \{S_k\} \leftarrow \mathrm{Plan}(x, o_t, \mathcal{S}_t)$
  for each subgoal g_k anchored by S_k **do**
    $\mathcal{B}^{app}_{S_k} \leftarrow \{f \in \mathcal{B}_{S_k} \mid \forall \phi \in \mathcal{P}_{S_k}, \phi(o_t) = \mathrm{true}\}$
    if $\mathcal{B}^{app}_{S_k} = \emptyset$ **then**
      $a_t \leftarrow \mathrm{ReAct}(o_t, g_k)$
    else
      $f \leftarrow \arg\max_{f \in \mathcal{B}^{app}_{S_k}} \mathrm{UtilityStat}(f)$
      $\theta \leftarrow \mathrm{Ground}(o_t, \Theta_{S_k})$
      $a_t \leftarrow f(\theta; o_t)$
    end if
    $o_{t+1} \leftarrow \mathrm{ExecuteOnEnv}(a_t)$
    if a_t is a skill invocation and $\mathcal{Q}_{S_k}(o_{t+1}) \neq \mathrm{true}$ **then**
      $\psi \leftarrow \mathrm{MatchFailure}(\mathcal{F}_{S_k}, o_{t+1})$
      if ψ exists and $\mathcal{R}_{S_k}[\psi]$ exists **then**
        $o_{t+1} \leftarrow \mathrm{ExecuteOnEnv}(\mathcal{R}_{S_k}[\psi])$
      end if
      $\mathcal{E}_{ref} \leftarrow \mathcal{E}_{ref} \cup \{(S_k, \psi, \mathrm{snap}(o_t, o_{t+1}))\}$
    end if
    $\tau \leftarrow \tau \oplus (a_t, o_{t+1})$
    if $\mathrm{TaskComplete}(o_{t+1}, x)$ **then**
      return $\tau, \mathcal{E}_{ref}$
    end if
  end for
end for
return $\tau, \mathcal{E}_{ref}$

in the main experiments. We report the average input+output token consumption per task instance and distinguish between the recurring action-generation cost and the overhead introduced by skill construction. For WebSync, this overhead consists of Skill Consolidation, which includes advice distillation, recipe formulation, constrained synthesis, and validation, and Recipe Refinement, which updates recipe contracts from observed execution failures.

As shown in Table 9, WebSync introduces additional overhead for skill consolidation and recipe refinement, resulting in a higher total token count for the initial run compared with the no-skill baseline. This is expected: unlike a memoryless agent that only generates actions, WebSync invests extra computation to convert successful and failed interaction traces into reusable, guarded skills.

However, this upfront investment yields a lower recurring inference cost. WebSync reduces the action-generation token consumption from 55,367.4 to 48,184.6 tokens, a reduction of approximately 13.0%. This reduction is consistent with the step-based results in the main paper: once a recipe-validated skill is available, the agent can invoke a compact executable procedure instead of repeatedly prompting the backbone model to reason through the same low-level browser operations.

The consolidation and refinement costs should therefore be interpreted as amortizable costs rather than costs paid in full for every future use. Once a recipe and its executable bindings are created, they are stored in the Skill Library and reused across later tasks with matching intent and page conditions. Similarly, Recipe Refinement is triggered only when runtime failures expose missing predicates, false-positive postconditions, or uncovered recovery cases. As the same recipes are reused over longer task streams, the average per-task cost converges toward the lower action-generation cost, making WebSync more cost-effective in settings with recurring web operations.

G Evolution Dynamics and Cold Start

To analyze how WebSync evolves as experience accumulates, we track cumulative performance on the WebArena Shopping domain, which contains 187 tasks and provides the longest single-domain sequence in WebArena. Table 10 reports cumulative success rate and average interaction steps at early checkpoints and later stages of the task stream.

The early-stage behavior of WebSync is close to the base agent because the Advice and Skill Libraries are still sparse, so few executable bindings pass the recipe precondition checks. After roughly 50–100 tasks, recurring Shopping patterns such as search, filtering, sorting, and cart updates are increasingly distilled into reusable recipes. As a result, the success-rate gap widens and the average number of interaction steps steadily decreases, indicating that WebSync gradually converts repeated low-level reasoning into reusable, guarded skills.

Library Growth. The library size is governed by reusable operation patterns rather than by the number of episodes. WebSync does not promote every successful trajectory into a new skill. Candidate Trace Selection retains only segments that satisfy recurrence, utility, and stability criteria. Recurring operations such as search, filtering, sorting, and cart updates are merged into shared Advice/Recipe/Skill entries, whose parameter schemas, preconditions, and postconditions are refined over time. As shown in Table 11, the final libraries remain much smaller than the task streams: WebArena has 655 tasks but only 64 advice entries

Method	Action Gen. Tokens	Skill Consolidation Tokens	Recipe Refinement Tokens	Total Tokens Tokens
Base (No Skills)	55,367.4	–	–	55,367.4
WebSync (Ours)	48,184.6	7,842.1	2,416.8	58,443.5

Table 9: **Token cost decomposition.** Action Gen. is the recurring inference cost for producing executable actions. Skill Consolidation and Recipe Refinement are amortizable overheads introduced by recipe-guided skill construction and failure-driven recipe updates.

After N Tasks	Cumulative SR (%)			Avg. Steps	
	Base	WebSync	ΔSR	Base	WebSync
10	60.0	60.0	+0.0	5.8	5.9
20	45.0	50.0	+5.0	6.7	6.1
30	50.0	53.3	+3.3	6.2	5.8
40	42.5	47.5	+5.0	6.5	5.9
50	36.0	40.0	+4.0	6.4	5.6
100	39.0	53.0	+14.0	6.2	4.3
150	38.7	61.3	+22.6	6.1	3.5
187	38.5	64.2	+25.7	6.1	3.1

Table 10: **Cold-start and evolution dynamics on WebArena Shopping.** Results are cumulative over the first N completed tasks. Early checkpoints are intentionally noisy because a small number of easy or hard tasks can move the cumulative success rate substantially.

Benchmark	# Tasks	Final # Advice	Final # Skills
WebArena	655	64	31
WebVoyager	643	118	57

Table 11: **Final library sizes.** WebSync stores reusable Advice/Recipe/Skill entries rather than appending every trajectory, so the final libraries remain much smaller than the number of task episodes.

and 31 skills, while WebVoyager has 643 tasks but 118 advice entries and 57 skills. This contrast also illustrates that growth depends more on task and interface diversity than on task count alone. WebVoyager induces a larger library despite having a similar number of tasks because it spans 15 live websites, requiring more site-specific bindings and page-condition variants. Therefore, library growth is primarily driven by reusable task families and interface patterns rather than the raw number of completed tasks.

Retrieval Overhead. At inference time, WebSync separates recipe-template retrieval from executable-binding selection. The agent first retrieves a small set of candidate Advice/Recipe templates using task intent and page-level semantic markers. DOM-level precondition checks are then applied only to the executable bindings attached to the retrieved recipes. If no binding passes these checks, the agent falls back to primitive actions while still using the retrieved advice as planning guidance. This two-stage design avoids treating the Skill Library as a flat cache that must be scanned exhaustively at every step.

Error Accumulation. WebSync limits error accumulation through two barriers. First, Recipe-Based Validation prevents synthesized tools from being committed unless their postconditions can be verified in matching contexts. Second, precondition guards prevent previously learned skills from firing in invalid states during inference. When a runtime failure exposes a missing predicate, false-positive postcondition, or uncovered recovery case, the failure is routed to Evidence-Based Recipe Refinement rather than blindly appending a new skill. Thus, failures are used to narrow or repair existing recipes, reducing the risk that stale or defective skills pollute the library.

H More Case Studies

In this section, we provide detailed examples of the synthesized skills and a step-by-step trajectory analysis to illustrate the robustness of WebSync.

H.1 Amazon Filtering Pseudocode

H.2 Skill Examples

To demonstrate the robustness of our framework, Figure 6 presents a qualitative comparison between skills synthesized by baseline methods and those generated by WebSync.

As shown in Figure 6(a), existing skill-centric approaches (e.g., PolySkill, ASI) often rely on naive trace cloning. The resulting code typically manifests as linear action sequences or high-level abstract wrappers. Such implementations are inherently brittle; they lack context awareness and error-handling mechanisms. Consequently, if the web environment exhibits asynchronous loading delays or unpredicted layout shifts, these rigid scripts fail to adapt, leading to immediate task termination.

Algorithm A.4 Skill: ROBUSTFILTERAMAZON

Require: *page* (Browser State), *target* (e.g., "Used - Good")
Ensure: SUCCESS or FAILURE with signal
Constraint: *Derived from Advice #AMZ-042 (DOM instability)*

```
function APPLYFILTER(page, target)
    // 1. Precondition (P): Guarded Execution
    if ¬ ISVISIBLE(page, "#s-refinements") then
        return FAILURE("Filter panel not loaded")
    end if
    // 2. Execution (B): Fast Procedure
    EXECUTE(page, "click_exact_match", target)
    WAIT(0.5s)
    // 3. Verification (Q): Ground Truth Check
    if VERIFYSTATE(page, target) is TRUE then
        return SUCCESS
    end if
    // 4. Recovery (R): Self-Correction
    LOG("Exact match failed. Triggering fuzzy recovery...")
    Strategy: Relax matching & target parent container
    EXECUTE(page, "click_fuzzy_parent", target)
    WAIT(1.0s)
    // Final Guard
    if VERIFYSTATE(page, target) is TRUE then
        return SUCCESS                          ▷ Recovered
    end if
    return FAILURE("Action failed after recovery")
end function
```

In contrast, Figure 6(b) illustrates how WEBSYNC leverages Skill Recipes as a governing contract to synthesize guarded executable procedures. Using an Amazon product filtering task as a case study, the synthesized skill follows the P-B-Q-R structure:

- **Preconditions (P): Resilient Entry Guards.** Instead of relying on fragile instantaneous checks (e.g., `is_visible`), the synthesized function employs robust synchronization primitives. It prioritizes `wait_for_selector` to handle network latency gracefully. Explicit recovery actions (e.g., `page.reload()`) are triggered *only* if the critical element fails to appear after a specific timeout, preventing unnecessary reloads on slow connections.

- **Verification (Q): State Confirmation.** Instead of assuming action success based solely on execution completion, the code explicitly verifies the outcome post-execution. In this case, it checks for expected URL parameter updates or specific DOM state changes to confirm the filter was applied.

- **Recovery (R) with Closed-Loop Validation.** The skill encapsulates verifiable self-correction logic. When the primary execution path (e.g., an exact-match click) fails due to DOM instability, the handler catches the exception and triggers a fallback strategy (e.g., a fuzzy click on the parent container). **Unlike baseline methods, WEBSYNC performs a second verification step after recovery.** If the state remains invalid even after the fallback action, the skill reports a failure, reducing the chance that the agent proceeds with a "false positive" success.

H.3 Example of executing web navigation tasks with WebSync

Figure 7 illustrates a qualitative comparison on an Amazon search task. The **Baseline Agent**, relying on atomic actions, exhibited two critical weaknesses: (1) *Navigational Instability*, getting trapped in a refresh loop due to SPA hydration issues (Steps 1–5); and (2) *Lack of State Verification*. While the agent attempted to click the "Used" filter selector, it failed to verify if the filter was successfully applied to the DOM. Consequently, the agent extracted mixed results (New and Used items) but incorrectly marked the task as complete, resulting in a **false positive**. In contrast, our **Recipe-Guided Agent** utilized a synthesized skill with explicit `postconditions` (e.g., asserting the presence of the active filter tag). This synchronized the browser state with the agent's intent before data extraction, achieving verifiable success in this example. Moreover, Figure 8, Figure 9 and Figure 10 provide a detailed trajectory comparison of the three approaches on a dynamic Amazon filtering task. The baseline agent relies purely on atomic actions, trapping it in a costly trial-and-error loop when facing asynchronous page loading. While the skill-centric baseline employs synthesized tools to reduce steps, the lack of execution contracts causes the tool to crash upon encountering unexpected layout shifts, forcing an inefficient fallback to low-level reasoning. In contrast, WebSync consolidates routine actions into more complex, compositional tools like `search_and_filter`. During the inference phase, WebSync leverages retrieved strategic advice to formulate a short-horizon plan and select the skill for its subgoals. Guided by preconditions and built-in recovery logic, the chosen tool handles dynamic UI exceptions internally and completes the task with fewer atomic trial-and-error actions.

System Prompt: Strategic Advice Extraction

You are a Senior Web Navigation Analyst. Your job is to analyze interaction traces to distill high-level

Input Data:

- **Task:** `{{task_description}}`
- **Execution Trace:** `{{trace_history}}` (Actions, Observations, and Outcomes)
- **Status:** `{{success_or_failure}}`

Analysis Objectives:

1. **Identify Bottlenecks:** Look for redundant steps, loops, or failed clicks caused by dynamic DOM elements (e.g., pop-ups, layout shifts).

2. **Generalize Patterns:** Abstract specific element IDs into semantic rules (e.g., instead of "Click div#123", suggest "Wait for the 'Sort' dropdown to be fully visible").

3. **Formulate Constraints:** Create explicit applicability conditions (σ) and verification checks (Q) for this type of task.

Output Format: Return a concise advice summary in the following JSON format:

```
{
  "trigger_condition": "When attempting to filter items...",
  "precondition_advice": "Ensure the overlay mask is gone.",
  "recovery_advice": "If click fails, try forcing a page reload."
}
```

Figure 3: The prompt used to distill raw interaction traces into reusable Strategic Advice ($\mathcal{M}_A$), serving as the cognitive foundation for skill synthesis.

System Prompt: Recipe-Guided Skill Synthesis (WebSync Section 3.4; Appendix A)

You are an expert Web Automation Engineer specializing in robust tool synthesis. Your goal is to [illegible] Skill Recipe contract.

Input Data:

- **Task Description:** `{{task_description}}`
- **Refined Skill Recipe S (JSON):** `{{refined_spec}}`
- **Strategic Advice (optional):** `{{advice_constraints}}`
- **Interaction Trace (optional):** `{{trace_history}}`

Synthesis Requirements (must satisfy the paper):

1. **Preconditions (P):** for EACH predicate in `{{refined_spec.preconditions}}`, add a guarded check (e.g., `assert ...`) before executing the core actions.

2. **Postconditions (Q):** for EACH predicate in `{{refined_spec.postconditions}}`, implement verification at the end; do not silently pass if any postcondition fails.

3. **Failure Signatures (F) + Recovery (R):** if verification fails, detect the failure mode using `{{refined_spec.failure_signatures}}` and run the corresponding `{{refined_spec.recovery}}` steps; retry the core logic at most once.

4. **Parameterization:** follow `{{params_schema}}` exactly; do not hardcode task-specific values.

5. **Defensive execution:** handle dynamic loading, popups/modals, and small UI shifts.

Implementation Notes (align with ToolLibrary style):

- Write an `async def` Python function whose name matches `{{refined_spec.bindings.tool_name}}` (or `{{refined_spec.name}}` if missing).
- First argument must be `page`.
- Prefer stable selectors and semantic markers; avoid brittle dynamic IDs.
- Use bounded retry logic; avoid infinite loops.
- Return a concise status string (include page title + URL if available) or raise a clear exception on failure.

Output: Return ONLY the Python code (no Markdown fences, no extra commentary).

Figure 4: The system prompt for Recipe-Guided Skill Synthesis.

System Prompt: Recipe-Informed Planning

You are a planning module. Given a user instruction and the current observation, you must produce a

Input:

- **Task instruction x:** `{{task_description}}`
- **Current observation o_t:** `{{observation}}`
- **Retrieved Recipes S_t (JSON list):** `{{retrieved_specs}}`

Your job:

1. Decompose the task into 1–5 subgoals.
2. For each subgoal, choose the BEST matching Recipe as an anchor (or "NO_SPEC" if none fits).
3. Prefer plans that are executable under the available capabilities (do not propose actions that have no supporting spec/tool unless necessary).

Output format (JSON ONLY): `{`

```
"plan": [
{
"subgoal": "...",
"spec_anchor": {
"name": "SPEC_NAME | NO_SPEC",
"match_rationale": "1-2 short reasons referencing sigma/page markers/intent"
}
}
]
}
```

Rules:

- Do NOT include any extra keys.
- Keep subgoals concrete and verifiable.

Figure 5: The system prompt for Recipe-Aware Planning.

Figure 6: **Qualitative Comparison of Skill Synthesis Approaches.** (a) **PolySkill & ASI.** generate abstract classes or linear action sequences without execution guards. (b) **WebSync (Ours)** synthesizes guarded skills with explicit preconditions (P), semantic execution (B), verification (Q), and recovery (R).

(a)PolySkill & ASI

```python
class AbstractShoppingSite:

    def search_product(self, query: str):
        """Searches for a product."""

    def add_to_cart(self, item_id: str, quantity:
     int):
        """Adds a specified item to the shopping
     cart."""

    ## --------------------
    ## Compositional Skills
    ## --------------------

    def find_and_add_to_cart(self, query: str,
     item_id: str):
        """Searches for a product and adds it to the
      cart."""
        # High-level abstraction only
        self.search_product(query)
        self.add_to_cart(item_id)
        self.checkout()
```

```python
def search_product(search_box_id: str, query: str):
    """
    Examples:
    search_product('595', 'sony bluetooth headphones
     ')
    """
    # Linear execution - brittle to async loading
    click(search_box_id)
    fill(search_box_id, query)
    keyboard_press('Enter')

def navigate_category(dropdown_id: str, category_id:
      str):
    """Navigate to a category via dropdown."""
    click(dropdown_id)
    click(category_id)
```

(b) WebSync (Ours)

```python
async def robust_search_filter(
    page, term, filter_txt
):
    # [P] Precondition: Guarded Entry
    # Logic: Wait first, reload only if truly stuck
    try:
        await page.wait_for_selector(
            "#twotabsearchtextbox",
            state="visible", timeout=3000
        )
    except TimeoutError:
        # Only trigger recovery if wait fails
        await page.reload()
        await page.wait_for_selector("#
     twotabsearchtextbox")

    # [B] Execution: Semantic Locators
    await page.fill("#twotabsearchtextbox", term)
    await page.press("#twotabsearchtextbox", "Enter"
     )

    # Wait handles async loading
    await page.wait_for_selector("div.s-main-slot")

    # [B] Execution: Robust Filtering
    try:
        # Fallback to list item containing text
        loc = page.locator(f"li:has-text('{
     filter_txt}')")
        await loc.first.click()
    except Exception:
        return f"Failure: Cannot locate '{filter_txt
     }'"

    # [Q] Verification: Strict State Check
    # Helper to verify server state (URL param)
    verify_func = "url => url.includes('
     p_n_condition')"

    try:
        await page.wait_for_function(
            verify_func, page.url, timeout=3000
        )
    except TimeoutError:
        # [R] Recovery: Self-Correction Strategy
        print("Click failed. Retrying parent span...
     ")
        await page.locator(
             f"span:has-text('{filter_txt}')"
        ).click()

        # [Critical] Re-Verification after Recovery
        try:
            await page.wait_for_function(
                verify_func, page.url, timeout=3000
            )
        except TimeoutError:
            # Algorithm 1: Line 22 (Fail if recovery
      fails)
            return "Failure: Action failed after
     recovery"

    return "Success: Filter Applied & Verified"
```

Our Approach (Recipe-Guided)

Task: Search "Switch Lite" with **"Used"** filter.

Summary: **Success in 2 Steps (12.8s)**.

- **Action:** `search_and_filter(..., filter="Used")`
- **Robustness:** The synthesized skill internally verified the DOM state change before returning control.
- **Obs:** *"Filter 'Used' clicked: True. Applied: True."*

Final Verdict: **True**. *Reason:* Valid result with confirmed filter state.

Baseline Agent (Atomic Actions)

Task: Search "Switch Lite" with **"Used"** filter.

Summary: **Failed after 9 Steps (70s)**.

- **Steps 1–5 (Instability):** Encountered SPA hydration issues (blank page). Wasted tokens on navigation retries.
- **Steps 6–8 (Interaction Error):** Attempted to click the "Used" selector. However, the agent **failed to verify** if the filter was actually applied (e.g., did not check for URL update or active badges).
- **Step 9 (Result):** Extracted mixed results (New & Used).

Final Verdict: **False**. *Reason:* **"Used" filter not checked.** The extracted list contained non-filtered items because the click action did not register correctly.

Figure 7: **Qualitative Comparison.** Left: Our approach successfully applies the "Used" filter using a synthesized robust skill. Right: The Baseline agent struggles with page stability and ultimately fails because it did not verify that the "Used" checkbox was actually selected.

Case Study: GPT-4o Base Trajectory

Task: Search "Xbox Wireless Controller" and apply "Green" and "4 Stars & Up" filters on Amazon.

[Deliberative Reasoning] **Step 0-1 (Initialization & Hydration Failure):**

- *Action:* Navigate to `https://www.amazon.com/`.
- *Observation:* Page appears empty (SPA not loaded). 0 links, 0 interactive elements.
- *Result:* Failure. Agent evaluates: "Encountered an empty page... Need to refresh." Refreshes page.

[Deliberative Reasoning] **Step 2-3 (Search Execution & Hydration Failure):**

- *Action:* Input 'Xbox Wireless Controller green' into search box (index 2396) and click 'Go' (index 3010).
- *Observation:* The search results page loads as an empty screen.
- *Result:* Failure. Agent evaluates: "The search was executed, but the page did not load correctly." Refreshes page.

[Deliberative Reasoning] **Step 4-5 (Filter Execution & Hydration Failure):**

- *Action:* Click 'Green' filter (index 19356) and '4 Stars & Up' filter (index 19129).
- *Observation:* The filtered results page loads as an empty screen.
- *Result:* Failure. Agent evaluates: "Encountered an empty page after refreshing the search results." Refreshes page.

[Deliberative Reasoning] **Step 6-7 (Extraction & Completion):**

- *Action:* Extract details of Xbox controllers matching the criteria and call `done`.
- *Result:* Success. Task completed, but required 7 steps and 3 manual recovery cycles.

Figure 8: An execution trajectory of a memory-less baseline agent (GPT-4o Base). The agent relies on atomic deliberative reasoning. While it eventually succeeds, it suffers from navigational instability, wasting multiple inference cycles (Steps 1, 3, and 5) to manually handle asynchronous UI loading (SPA hydration issues).

Case Study: Skill-Centric Baseline (e.g., SkillWeaver) Trajectory

Task: Search "Xbox Wireless Controller" and apply "Green" and "4 Stars & Up" filters on Amazon.

[Tool Execution] Step 1 (Search Strategy):

- *Action:* Invoke synthesized tool `search(query="Xbox Wireless Controller green")`.
- *Observation:* The search executes smoothly and the initial results page loads.
- *Result:* Success.

[Tool Execution] Step 2 (Filter Strategy & Unverified Failure):

- *Action:* Invoke synthesized tool `apply_filter(conditions=["Green", "4 Stars & Up"])`.
- *Tool Internal Execution:* The script mechanically attempts to click the filter selectors based on cached DOM paths.
- *Observation:* The Amazon page enters an asynchronous SPA hydration state (blank page) during the click.
- *Result:* Exception Thrown (ElementNotInteractable).

[Atomic Fallback] Step 3-7 (Trial-and-Error Reasoning):

- *Agent Reflection:* "The filter tool failed. I need to manually locate and apply the filters."
- *Step 3:* Agent refreshes the page to escape the blank state.
- *Step 4:* Agent manually locates the 'Green' filter (index 19356) and issues a primitive `click` action.
- *Step 5:* Page goes blank again (hydration). Agent evaluates failure.
- *Step 6:* Agent refreshes the page again.
- *Step 7:* Agent manually locates and clicks '4 Stars & Up' filter (index 19129).

Figure 9: An execution trajectory of a baseline skill-centric agent (e.g., SkillWeaver). The absence of robust recipes (preconditions and recovery handlers) causes the synthesized `apply_filter` tool to crash upon encountering a dynamic layout shift. Consequently, the agent is forced to revert to expensive, atomic trial-and-error reasoning.

Case Study: WebSync (Ours) Trajectory

Task: Search "Xbox Wireless Controller" and apply "Green" and "4 Stars & Up" filters on Amazon.

[Recipe-Informed Planning] Step 1 (Advice Retrieval & Plan Generation):

- *Context:* Agent observes the Amazon homepage and analyzes the intent.
- *Retrieval:* Retrieves Strategic Advice (`#AMZ-042`): "Amazon filtering is brittle due to SPA hydration and unclickable overlays. Avoid atomic ReAct clicks; mandate guarded filter tools."
- *Plan Decomposition:* Agent generates a short-horizon plan with bounded skill anchors:
  - **Subgoal 1:** Execute search and apply required filters. → *Skill Anchor:* `robust_search_filter`
  - **Subgoal 2:** Extract the compliant product list. → *Skill Anchor:* `extract_results`

[Guarded Tool Execution] Step 2 (Unified Search & Filter):

- *Action:* Invoke mapped tool `robust_search_filter(term="Xbox Wireless Controller green", filter_txt=["Green", "4 Stars & Up"])`.
- *Tool Internal Execution (Contract Enforcement):*
  - **[P] Precondition:** Explicitly waits for `#twotabsearchtextbox` and filter panel to be fully interactive, gracefully bypassing the blank hydration state.
  - **[B] Execution:** Inputs query and executes click events using semantic locators.
  - **[R] Recovery:** Detects a layout shift during the 'Green' filter click; autonomously triggers a fuzzy click on the parent container.
  - **[Q] Verification:** Asserts URL parameters (`p_n_condition`) and active DOM tags.
- *Result:* Success. Browser state is verified and synchronized with the applied filters.

[Guarded Tool Execution] Step 3 (Extraction & Completion):

- *Action:* Invoke mapped tool `extract_results(target_rating=">4")` and call `done`.
- *Result:* Success. Task completed efficiently in 3 high-level steps.

Figure 10: An execution trajectory of WebSync. Unlike baseline methods, WebSync begins with a Recipe-Informed Planning phase, retrieving strategic advice to avoid known pitfalls (e.g., SPA hydration) and anchoring subgoals to specific Skill Recipes. It then delegates execution to guarded tools equipped with P-B-Q-R contracts, handling dynamic UI issues internally and completing the task with fewer atomic trial-and-error actions.