Skill Induction for Code Agents on Web Automation

Demi Wang
Carnegie Mellon University
Pittsburgh, PA, USA
demiw@andrew.cmu.edu

Lintang Sutawika
Carnegie Mellon University
Pittsburgh, PA, USA
lsutawik@andrew.cmu.edu

Graham Neubig
Carnegie Mellon University
Pittsburgh, PA, USA
gneubig@andrew.cmu.edu

Abstract

Skill induction has been widely studied as a way for web agents to accumulate reusable knowledge across tasks, but almost entirely under action-based frameworks where skills remain bound to a particular scaffolding. We instead study skill induction on code-native web agents, where skills take the form of standalone Playwright functions rather than compositions over a fixed action vocabulary. On WebArena-Verified, moving to a code-native substrate changes the skill-induction picture in three connected ways. First, a hybrid code agent that combines browser-tool navigation with Playwright script execution sets a substantially stronger no-skill baseline, outperforming a BrowserGym action-based agent on the same backbone model by 10.3 pp on the 104-task subset. Second, on this stronger baseline naive skill induction can actively degrade performance and prompt-level self-verification offers only marginal correction; we trace this to confirmation bias and address it with a multi-agent pipeline that decouples solving, verification, and updating into independent stages, admitting skills only through a gated mechanism. The full pipeline reaches 67.2%, a 6.4 pp improvement over the no-skill code agent at fixed skill format, and 10.3 pp above BrowserGym + ASI on the same 104-task subset. Third, the induced skills are standard Python functions and portability follows naturally from the representation: a preliminary case study shows that the same skill files drop into a different agent framework via directory copy, with no adaptation step.

CCS Concepts: • **Computing methodologies** → **Intelligent agents**.

Keywords: web agents, code agents, skill induction, web automation

1 Introduction

Skill induction has been widely studied as a way to improve web agents by accumulating reusable knowledge across tasks. This line of work has developed almost entirely within action-based frameworks, where agents choose from a fixed vocabulary of primitive browser operations such as click, type, and scroll, and observe web-page changes through accessibility trees or screenshots. This approach tightly couples both the agent's behavior and the skills it learns to a particular scaffolding's action schema, observation format, and state tracking.

Yet despite the many variations, many of these primitives operate as thin wrappers around Playwright,[1] a browser-automation library that coding agents can interface with directly. The two interaction modes differ in expressiveness, rather than in what they ultimately drive. An action-based agent can issue a click or a fill on a single accessibility-tree element, but it cannot write a loop over table rows, run a CSS selector to extract specific fields, or execute JavaScript to read data that is not exposed in the accessibility tree. A code-native agent can do all of these directly through Playwright. Coding agents have in turn been shown to achieve increasingly strong performance on a wide variety of tasks by programming solutions [2, 17].

To handle novel-yet-recurring tasks without the cost of further parametric training, agents are also increasingly supplied with task-specific instructions. A growing line of work has explored how such *skills* can be induced by the model through interaction over time, with agents building up reusable knowledge as they gain experience [15, 16, 19, 20]. These methods, however, still operate within action-based frameworks and produce skills composed of primitive browser operations that remain bound to a specific scaffolding.

In this work, we study skill induction on code-native web agents, where skills take the form of standalone Playwright functions: executable, composable, and grounded in standard browser APIs rather than any particular agent framework. Moving to a code-native substrate turns out to change the skill-induction picture in three connected ways, which together form the contributions of this paper.

The first change is that the no-skill baseline becomes substantially stronger. A hybrid code agent that combines browser-tool navigation with Playwright script execution outperforms a BrowserGym action-based agent on the same backbone model by 10.3 pp on the WebArena-Verified 104-task subset, before any skill mechanism is invoked. To our knowledge this is the first study of online skill induction in this setting, and it raises the bar that any skill-update policy must clear.

The second change is that this stronger baseline exposes the limits of existing induction strategies. Naively admitting every induced skill to the library can actively degrade performance, collapsing Shopping success from 65.5% to 27.6% as faulty abstractions propagate across tasks. Prompt-level

[1] https://playwright.dev/: a cross-browser automation library originally developed for end-to-end testing and debugging of web pages.

self-verification provides only marginal correction, because the solving agent tends to trust its own prior reasoning over contradictory page evidence. We trace this confirmation bias to a single missing ingredient, context isolation between solving and verification, and address it with a multi-agent pipeline that decomposes the skill lifecycle into solve, verify, and update stages, with a gated update mechanism that admits skills to the library only when an isolated verifier and a separate update agent both agree. The full pipeline reaches 67.2% on the same subset, a 6.4 pp improvement over the no-skill code-agent baseline at fixed skill format, and 10.3 pp above BrowserGym + ASI on the aligned subset. Appendix C reports paired McNemar tests for the claim-aligned comparisons.

The third change is a property the previous regime simply does not have. Because our induced skills are standard Python functions calling public Playwright APIs, they are not bound to the framework that produced them. In a preliminary case study (Section 4.5), the same skill files drop into the workspace of a generic Claude Code subagent via directory copy, with no adaptation step, and the subagent succeeds on three WebArena tasks across three domains that it cannot solve unaided. Action-level skills, by construction, do not transfer this way.

Together, these three changes suggest that on a code-native substrate the binding constraint of online skill induction shifts from skill generation to verification, while portability follows naturally from the representation.

2 Related Work

Web agents and code-based interaction. Web navigation agents have been widely studied across diverse task domains including e-commerce, social forums, software development, and content management [4, 8, 21]. Most agents operate within action-based frameworks such as BrowserGym [3], selecting from primitive browser operations and observing pages through accessibility trees. Beyond Browsing [12] showed that API-based and hybrid agents outperform pure browsing agents on WebArena, and coding agent frameworks such as OpenHands [14] have demonstrated strong performance on software tasks through code generation and execution. We extend this direction to web automation, adopting a hybrid code agent that combines browser tool navigation with Playwright script execution as the foundation for skill induction.

Skill induction for web agents. Recent surveys frame agentic skills as reusable procedural capabilities with explicit applicability conditions, execution policies, and interfaces, and organize the field around the skill lifecycle of discovery, verification, storage, and update [7, 18]. Within this lifecycle, prior work has explored several skill representations. AWM [16] induces natural language workflows from successful episodes; SteP [10] provides human-written procedural guidance. ASI [15] represents skills as executable action macros that compose primitive browser operations, verified through trajectory re-execution. SkillWeaver [20] discovers skills through autonomous offline exploration, honing them into Playwright-based APIs without test-time queries. PolySkill [19] decouples a skill's abstract goal from its website-specific implementation through polymorphic abstraction.

We differ from prior code-skill methods on three axes. ASI's trajectory-replay verification is functional and admits skills that execute successfully but produce incorrect results, the exact failure mode our Shopping Admin case studies (Appendix A.2) target. SkillWeaver operates offline and is complementary to our online setting. PolySkill's polymorphic abstraction is orthogonal to the gating mechanism we propose. More broadly, all of these methods operate within action-based agent frameworks; our skills are standalone Playwright functions in the same language the agent already uses, inheriting verifiability and composability from code while remaining structurally portable across agent frameworks.

Skill quality and evaluation. SkillsBench [9] reports that curated skills boost pass rates by 16.2 pp on average while self-generated skills provide negligible benefit and hurt in nearly a third of tasks. We provide a mechanistic account of this gap on a stronger substrate, showing that on hybrid code agents naive skill induction is actively harmful, and identifying confirmation bias and abstraction drift as the underlying failure modes (Section 4). ASI addresses skill quality through execution-based verification by re-running skills and checking for exceptions, but this functional check misses skills that execute successfully yet produce incorrect results. We target this gap directly with semantic verification: an independent agent cross-validates skill outputs against live page evidence, and a gated update mechanism prevents low-quality skills from entering the library.

3 Method

3.1 Code Agents for Web Automation

Most web agents interact with browsers through action-based frameworks such as Browser Use [1] and BrowserGym [3], which expose a fixed vocabulary of high-level operations (e.g., click, type, scroll) over a structured page representation such as an accessibility tree. The agent can observe the page structure, but can only act through predefined operations on individual elements. It cannot, for instance, write a loop to iterate over table rows, run a CSS selector to extract specific fields, or execute JavaScript to access data not exposed in the accessibility tree.

Recent work [5, 11] has demonstrated the value of combining GUI interaction with programmatic code execution for

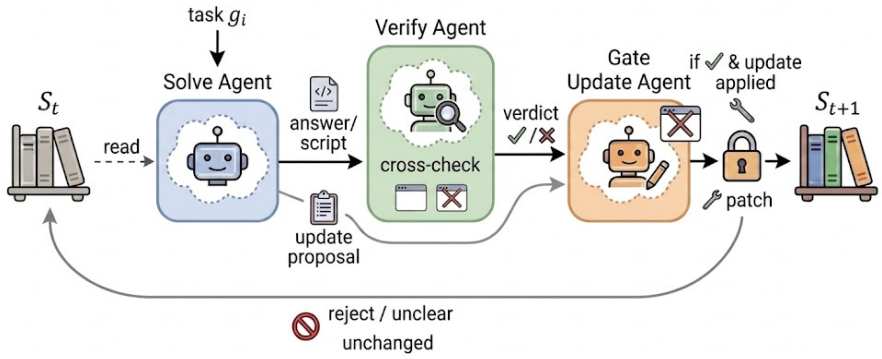

Figure 1. Multi-agent skill induction pipeline. The Solve Agent reads the current skill library $\mathcal{S}_t$ and produces an answer, script, and candidate update proposal. The Verify Agent runs in an isolated context (no access to the solve trajectory) and cross-checks the script output against live page evidence. Only when the verdict is `accept` *and* the Update Agent applies a patch does the library advance to $\mathcal{S}_{t+1}$; otherwise the library is left unchanged. The gate is asymmetric: false rejects forgo an update opportunity, while false accepts are the only path for a faulty skill to enter the library.

desktop automation. However, these operate at the OS level—generating pixel-coordinate mouse actions or file-system commands—without direct access to application-internal structure. In web automation, a richer option is available: Playwright scripts can operate directly on the browser's DOM, enabling structured queries, programmatic data extraction, and JavaScript evaluation within the page context.

We build on OpenHands [14], an open-source code agent framework, and equip it with two tools:

Browser tool. The browser tool [13], built on Browser Use [1], supports action-level web navigation, including following links, reading page content, filling out forms, and clicking elements via an accessibility-tree representation. Like other action-based frameworks, it is implemented on top of Playwright.

Terminal tool. A sandboxed shell for executing arbitrary code. The agent can write and run Playwright scripts to interact with the same browser programmatically, such as querying elements via CSS selectors, extracting structured data from tables, handling pagination with loops, and executing in-page JavaScript to access data not available through the accessibility tree.

The agent has simultaneous access to both interfaces and dynamically selects between them based on task requirements: the browser tool for navigation and page understanding, and Playwright scripts via the terminal for precise data extraction and complex multi-step operations. We evaluate browser-only and code-only configurations as ablations in Section 4, and compare against action-based agents as a baseline.

3.2 Problem Setup: Online Skill Induction

We formalize web automation as a sequential decision-making problem. Given a natural language goal g and a set of starting URLs, an agent must interact with live websites to complete the task. The transition dynamics are governed by external web environments and tool execution, and are not known to the agent. This setting can be viewed as a partially observable Markov decision process (POMDP) with unknown transition dynamics.

State and observation. The environment state $s_t = (s_t^{\text{browser}}, s_t^{\text{fs}})$ consists of the current browser state (DOM tree, page content, and navigation history) and the agent's file system workspace. The agent receives partial observations $o_t = O(s_t)$: the browser tool returns a structured accessibility tree of the current page, and the terminal returns stdout/stderr of executed code. The workspace s_t^{fs} is fully visible to the agent as an ordinary file system, containing task-specific code, output files, and the skill library.

Action space. At each step, the agent issues a tool call $a_t = (k_t, c_t)$, where $k_t \in \{\textsc{Browser}, \textsc{Terminal}\}$ selects the tool and c_t is the tool-specific input (a browser action command or an executable code snippet). The browser tool exposes a finite vocabulary of operations over indexed accessibility tree elements, including navigate, click, type, scroll, and read page state. The terminal tool accepts arbitrary code execution, making the action space effectively open-ended and programmatic. The agent policy π_L, instantiated by an LLM, selects the tool and generates the input based on the current observation, the full interaction history, and the current goal g:

$$\pi_L(g,\ o_{0:t},\ a_{0:t-1};\ \mathcal{S}) \ \rightarrow\ a_t = (k_t,\ c_t) \tag{1}$$

Table 1. Skills induced for the same task (*"Change my bio to 'I am a robot' on the discussion forum,"* Reddit, task 399). ASI composes BrowserGym primitives using hard-coded accessibility-tree element IDs (e.g., `'155'`, `'116'`); our skills use semantic CSS selectors (e.g., `a[href=...]`, `button:has-text(...)`) that generalize across page layouts.

ASI: Action-Composed Skill

```
def edit_and_save_bio(
    edit_bio_id, bio_textbox_id,
    save_button_id, new_bio):
    """Edit and save the user biography.

    Examples:
        edit_and_save_bio(
            '155', '116', '250',
            'I am a robot')
    """
    click(edit_bio_id)
    fill(bio_textbox_id, new_bio)
    click(save_button_id)
```

Ours: Code-Native Skill

```
async def edit_user_biography(
    page: Page,
    username: str,
    new_bio: str) -> bool:
    """Edit the user's biography."""
    await page.click(
        f'a[href="/user/{username}'
        f'/edit_biography"]')
    await page.fill(
        'textarea', new_bio)
    await page.click(
        'button:has-text("Save")')
    return True
```

where $\mathcal{S}$ denotes the current skill library (defined below). The agent continues until it writes a structured response file or reaches a predefined horizon $H_{\max}$. Each episode corresponds to a goal g_i and produces an interaction trace $\tau_i = (o_{0:H_i}, a_{0:H_i-1})$ where $H_i \leq H_{\max}$. Each episode yields a binary reward $r(\tau_i, g_i) \in \{0, 1\}$ determined by a task-specific functional evaluator. This setting differs from standard reinforcement learning in that adaptation occurs through explicit modification of an external skill library rather than updating policy parameters.

Skill library. The skill library $\mathcal{S}$ is stored within the workspace s_t^{fs} as a directory of Python modules organized by website domain (e.g., `skill_library/shopping.py`). Each skill is a parameterized Python function with docstrings, type hints, and usage examples. Skills are executable code modules that can be invoked via the terminal tool and may produce side effects on both the browser state and the file system. All skills are directly visible and importable without requiring an explicit retrieval module: the agent can browse the skill directory, read source code to judge applicability, and call functions via standard Python imports. Unlike prior work where skills are injected into the prompt by a retrieval module [20] or added to a framework-managed action space [15], skills in our setting are first-class code objects that the agent uses through the same mechanisms as any other Python library.

Online skill induction. We consider a sequential setting where goals $G = \{g_1, \ldots, g_N\}$ arrive in order. After each episode, the skill library may be updated: $\mathcal{S}_t \rightarrow \mathcal{S}_{t+1}$. The objective is to maximize cumulative task success over the sequence through skill accumulation. How this update is performed, whether skills are admitted unconditionally or filtered through a verification gate, is the central question we address in the next section.

3.3 Multi-Agent Skill Induction Pipeline

A natural approach to online skill induction is to let the solve agent induce and verify skills within the same conversation. However, we find that naive skill induction, where all induced skills are admitted to the library without verification, can degrade performance even on strong baselines. Adding prompt-level self-verification instructions improves over the naive approach but provides only marginal gains, as the agent tends to trust its own prior reasoning over contradictory page evidence. We refer to this failure mode as *confirmation bias* and analyze it quantitatively in Section 4.

These observations motivate a multi-agent design that decomposes the skill lifecycle into three independent stages, each executed by a separate agent with its own conversation context. The agents communicate exclusively through structured artifacts written to the shared workspace. Figure 1 illustrates the overall pipeline.

Solve Agent. The solve agent receives the task goal g, the current skill library $\mathcal{S}_t$, and the workspace. It executes the task using the hybrid tool set, optionally importing and calling existing skill functions. Upon completion, it produces three outputs: (1) a structured response file (`agent_response.json`) containing the answer or confirmation of task completion; (2) a Playwright script (`final_automation.py`) encoding the procedure used to obtain the submitted answer, which the verify agent will independently execute; and (3) a candidate skill update file recording the agent's recommendation on whether skills should be added or modified, including target files and update type. The solve agent is instructed not to modify the skill library directly. The library is read-only during the solve phase, ensuring that unverified changes cannot leak into $\mathcal{S}_t$.

Verify Agent. The verify agent runs in a separate conversation with its own browser context. It has no access to the

solve agent's reasoning history and receives only the task goal, login credentials, and the shared workspace containing the solve agent's outputs. The verify agent follows a cross-validation protocol: (1) it navigates to the relevant pages independently using the browser tool, employing a different method than the solve agent's script (e.g., using built-in search or filters rather than DOM scraping) to determine the correct answer; (2) it runs `final_automation.py` exactly once, without debugging or modification, and records the output; and (3) it compares its independent browser evidence against the script output and the submitted answer. If the browser evidence disagrees with the submitted answer, the verdict must be `reject`, regardless of how small the discrepancy appears. Exact answers are required for counts and specific values, and the agent must not rationalize discrepancies. The output is a verdict $v \in \{\texttt{accept}, \texttt{reject}, \texttt{unclear}\}$ with supporting evidence. This design addresses confirmation bias by construction: the verify agent has never seen the solve agent's reasoning trajectory, so it cannot be anchored to prior conclusions.

Update Agent. The update agent runs only when two conditions are met: the verify agent's verdict is `accept` and the solve agent's candidate update recommends a skill modification. It operates on a clean copy of the skill library restored from the version-controlled repository, rather than the solve agent's workspace copy. The update agent reads the verify agent's verdict and evidence, the solve agent's candidate update, and the automation script, then applies targeted fixes to the relevant skill functions. It produces a structured output recording whether an update was applied, which files were modified, and the reason. The update agent does not have access to the browser and works only with the workspace files, ensuring that updates are grounded in evidence from the prior stages rather than independent re-exploration.

Gated Skill Sync. Skills are committed to the library only when both the verify and update stages confirm quality:

$$\mathcal{S}_{t+1} = \begin{cases} \mathcal{S}' & \text{if } v = \texttt{accept} \ \wedge \ \texttt{applied\_update} = \texttt{true} \\ \mathcal{S}_t & \text{otherwise} \end{cases} \tag{2}$$

where $\mathcal{S}'$ is the updated library produced by the update agent. All other combinations, including `reject`, `unclear`, or the update agent declining to apply changes, leave the library unchanged. After a successful update, changes are committed to the version-controlled repository, ensuring they persist for subsequent tasks. This gate is asymmetric by design: false rejects are safe, as the library simply misses an update opportunity, while false accepts represent the only path through which a faulty skill can enter the library. The gated mechanism minimizes this risk by requiring independent agreement between two agents that have never shared context.

4 Experiments

4.1 Setup

Benchmark. WebArena [21] is a realistic web benchmark consisting of 812 tasks across self-hosted website domains. Tasks are specified by natural language intents and require agents to navigate, retrieve information, or modify state on live web applications. We evaluate on WebArena-Verified [6], a re-audited release that replaces the original brittle string matching and LLM-as-judge evaluators with deterministic scoring: retrieval tasks are evaluated via data-type-aware JSON matching, and state-changing tasks are verified through HAR-based network event checks. We evaluate on four website domains: GitLab (software development), Reddit (social forum), Shopping (e-commerce), and Shopping Admin (content management).

We exclude Map tasks because WebArena-Verified's exact-match evaluator systematically rejects continuous-valued outputs (coordinates, distances, travel times) where minor formatting differences produce false negatives unrelated to agent capability. A manual audit of all 12 Map tasks under a comparable code-agent configuration confirmed that the dominant failure modes are evaluator-side rather than agent-side, including address-format mismatches, sub-minute discrepancies in route times from coordinate rounding, and pure unit-formatting differences. Including Map would only add noise to substrate and skill comparisons. We separately exclude multi-domain tasks to focus on within-domain skill accumulation.

Agent configurations. We compare the following configurations, all using Claude Sonnet 4 as the backbone LLM:

- **BrowserGym (action-based)**: an agent operating within BrowserGym [3], selecting from a fixed vocabulary of primitive operations (click, fill, scroll, navigate, etc.) over accessibility tree elements. This represents the standard paradigm used by prior skill induction methods.
- **BrowserGym + ASI**: the same action-based agent augmented with ASI [15], an online skill induction pipeline that composes BrowserGym primitives into reusable programmatic skills and verifies them by replaying skill-invoking trajectories and checking task success.
- **Code agent (no skill)**: our hybrid code agent (subsection 3.1) with both browser tool and terminal access, but no skill library.
- **Code agent + multi-agent skill**: our full system with the multi-agent skill induction pipeline (Section 3.3). The skill library is initialized empty and skills accumulate sequentially in task ID order within each domain.

Ablation settings. We evaluate on a 104-task subset (constructed proportionally from the four domains: `shopping=29`, `shopping_admin=27`, `gitlab=18`,

Table 2. Task success rate (%) on WebArena-Verified (104-task subset). Avg is the unweighted mean across domains.

Setting	Shop	Admin	GitLab	Reddit	Avg
BrowserGym	58.6	44.4	72.2	26.7	50.5
BrowserGym + ASI	48.3	59.3	83.3	36.7	56.9
Code agent (no skill)	65.5	55.6	88.9	33.3	60.8
Code agent + skill (ours)	**65.5**	**66.7**	83.3	**53.3**	**67.2**

Table 3. Tool configuration ablation on WebArena-Verified subset.

Setting	Shop	Admin	GitLab	Reddit	Avg
Playwright-only	27.6	18.5	44.4	23.3	28.5
Browser-only	37.9	**63.0**	77.8	**50.0**	57.2
Hybrid (both)	**65.5**	55.6	**88.9**	33.3	**60.8**

reddit=30). All main-table results (Table 2) and both ablation studies (tool configuration in Table 3 and skill pipeline in Table 4) are reported on the same 104-task subset, so all comparisons are like-for-like.

Metrics. We report task success rate (binary) per domain and averaged across domains. In our analysis, we additionally examine per-task regression and improvement counts relative to the no-skill baseline, skill adoption patterns and their correlation with task outcomes, and verification gate statistics to characterize the reliability of the quality control mechanism.

4.2 Main Results

Table 2 shows task success rates across four WebArena-Verified domains.

Code agents outperform action-based agents. Without any skill library, the hybrid code agent achieves 60.8% average success rate, surpassing the BrowserGym action-based agent (50.5%) by 10.3 pp. The code agent outperforms BrowserGym on all four domains, with the largest gains on GitLab (+16.7 pp) and Shopping Admin (+11.2 pp), and smaller but consistent gains on Shopping (+6.9 pp) and Reddit (+6.6 pp). This confirms that code-level browser interaction provides a stronger foundation than fixed action vocabularies for web automation.

Multi-agent skill pipeline provides further gains. Adding the multi-agent skill induction pipeline improves the code agent from 60.8% to 67.2% (+6.4 pp). The strongest gains appear on Reddit (+20.0 pp) and Shopping Admin (+11.1 pp), domains where the agent benefits from accumulated procedural knowledge (e.g., API-endpoint mappings for UI actions on Reddit, pagination-aware table iteration on Shopping Admin). Shopping remains flat (65.5%) despite frequent per-task regressions that tend to cancel out in aggregate, consistent with contract-shaped failures. GitLab shows a slight decrease (88.9% to 83.3%): the baseline is already very high, leaving little room for skill-based improvement while introducing risk of regression from irrelevant or misapplied skills that interfere with otherwise correct trajectories. Detailed per-domain analysis is provided in Section 4.4.

Two independent sources of gain. We deliberately separate the two contributions rather than reporting a single 67.2% vs. 56.9% headline, since that comparison conflates substrate change (BrowserGym → code agent) with skill-update policy (ASI's trajectory-replay verification → our gated multi-agent pipeline). The substrate effect is isolated by the no-skill rows: code agent (60.8%) over BrowserGym (50.5%) is +10.3 pp without invoking any skill mechanism. The verification-gating effect, i.e., preventing erroneous skill updates via an isolated verifier, is isolated by Table 4, which holds skill format fixed (Playwright functions in every skill row) and reports +6.4 pp from no-skill (60.8%) to multi-agent (67.2%). ASI's +6.4 pp gain over BrowserGym is similar in magnitude to ours but starts from a substantially weaker baseline; the higher absolute ceiling we reach is attributable to the substrate, while the policy that lets us *climb* that ceiling is the gated pipeline.

4.3 Ablation Studies

Tool configuration. Table 3 compares the three tool configurations without skill libraries. Playwright-only performs worst across all domains (18.5–44.4%), as writing correct automation scripts without live browser feedback is difficult. Browser-only is stronger (37.9–77.8%) but varies widely: it excels on GitLab (77.8%) and Shopping Admin (63.0%) where tasks benefit from structured accessibility tree navigation, but struggles on Shopping (37.9%) where complex data extraction requires programmatic DOM access. The hybrid configuration matches or exceeds the better of the two on Shopping and GitLab; on Shopping Admin the browser-only setting is slightly higher (63.0 vs. 55.6), and on Reddit it is highest (50.0 vs. 33.3) because the network-event evaluator penalizes the hybrid agent's UI clicks that record empty HARs (Appendix A.3). On the macro average the hybrid configuration is still strongest (60.8 vs. 57.2), confirming that the two tools provide complementary capabilities and that simultaneous access to both is important for robust performance.

Skill pipeline. Table 4 isolates the contribution of each component in our skill pipeline. Without verification, naive skill induction severely degrades performance: Shopping drops from 65.5% to 27.6% as buggy skills propagate across tasks. This confirms that verification is necessary but insufficient without proper isolation.

Adding verification instructions to the solve agent's prompt (self-verification) prevents the worst regressions

Table 4. Skill pipeline ablation on WebArena-Verified (104-task subset).

Setting	Shop	Admin	GitLab	Reddit	Avg
No skill	65.5	55.6	88.9	33.3	60.8
Naive skill	27.6	55.6	66.7	53.3	50.8
Self-verification	65.5	48.1	83.3	53.3	62.6
Multi-agent (ours)	65.5	**66.7**	83.3	**53.3**	**67.2**

but introduces a different failure mode: Shopping Admin drops from 55.6% to 48.1% due to confirmation bias, where the agent discovers discrepancies between script output and page evidence but trusts its own prior reasoning. Reddit benefits (+20.0 pp) because skill value in this domain comes from accumulated content (API-endpoint mappings for UI actions) that does not require gating to harvest.

Our multi-agent pipeline addresses confirmation bias by isolating verification into a separate agent with no access to the solve agent's reasoning. This recovers Shopping Admin to 66.7% and achieves the strongest overall average at 67.2%, a +4.6 pp gain over self-verification. The improvement demonstrates that context isolation is the critical mechanism, not the verification instructions themselves.

4.4 Analysis

4.4.1 Skill value is domain-dependent. The +6.4 pp average gain hides large per-domain spread: Reddit **+20.0 pp**, Shopping Admin **+11.1 pp**, Shopping flat, and GitLab **−5.6 pp**. The pattern aligns with how much repeatable procedural structure each domain offers. On Reddit, multi-agent flips 7 tasks from fail to pass over the no-skill baseline (against 1 regression). Case inspection across the broader skill-aware Reddit regime (Appendix A.3) presents several representative cases, which cluster into two recurring mechanisms that account for these wins: skills teach the agent which API endpoint corresponds to a UI mutation, repairing tasks where the no-skill agent records empty network events (upvote, comment, post submission), and skill docstrings act as semantic frameworks that nudge live re-navigation over stale hard-coded observations on extraction tasks. Shopping Admin gains follow a more procedural pattern: pagination-aware helpers iterate the full admin table while the no-skill agent reads only the first page. By contrast, GitLab starts at 88.9% under no-skill—little headroom to absorb any regression—and Shopping tasks frequently turn on contract details (units, empty-result protocol, food subtotal vs. order total) that resist procedural reuse, so a plausible abstraction can still yield the wrong answer form.

4.4.2 Naive skill induction propagates abstraction drift, not just bugs. Naive skill (no verification) lowers the no-skill macro average from 60.8% to 50.8% on the 104-task subset (Table 4). This aggregate number hides a strong domain split: on Reddit alone, Naive skill matches Self-verification and Multi-agent at 53.3% (+20.0 pp over no-skill), indicating that Reddit's gain comes from library content (API-endpoint knowledge for UI mutations) rather than from verification gating. The collapse is concentrated on Shopping, where 12 of 29 tasks flip from pass to fail. Inspection of the version-controlled skill repository surfaces *pollution chains* in which one false accept poisons multiple downstream tasks: a single failed task adds a wrong helper to the library, the next task imports the helper and fails, and a third task layers more code on top of the wrong abstraction (full chains in Appendix A.1). The failure mode is therefore not merely "the function has a bug"; it is that an unverified abstraction reframes how downstream solves perceive the task, and once the wrong frame is in place the trajectory remains locally coherent but globally wrong.

4.4.3 Self-verification suffers from confirmation bias on derived computations. Adding self-verification recovers Shopping's catastrophic collapse (back to 65.5%) but Shopping Admin drops 7.5 pp because the same agent now both writes and judges its own derived computation, and tends to trust the script over the page. The recurrent qualitative pattern is that the agent inspects the relevant page region, sometimes even notices contradictory evidence (e.g. a pagination footer reading "5 records found" next to a script result of 10), but still keeps the script result because it treats its own computation as more authoritative than the page. In other cases the agent writes an approximate sampling heuristic when an exact existing skill and a direct page read are both available. These are not failures to *look* at the page; they are failures to let page evidence override prior reasoning. We document five concrete Shopping Admin cases of this confirmation-bias pattern in Appendix A.2.

4.4.4 Multi-agent decomposition rescues confirmation-bias failures. Replacing prompt-level self-checking with an isolated verify agent (no access to the solve trajectory) recovers Shopping Admin from 48.1 to 66.7. The recovery is clean on counting tasks where self-verify trusted its own script, but ranking and aggregation tasks remain hard because their failure is *contract-shaped* rather than *script-shaped* (Appendix A.2 and A.4). The pipeline is not a complete fix. On the multi-agent runs over the 104-task subset (99 tasks with complete pipeline artifacts; 31 failures), **20/31 = 64.5% of failed tasks still received `verify=accept` and 13/31 = 41.9% still wrote updates back to the library** (Appendix B). The residual failures fall into two patterns. The first is an *evaluator-contract gap*: the verifier confirms that the page state is correct, but the answer fails the evaluator's required format or network-event contract (`project id` vs. `project path`, `NOT_FOUND_ERROR` vs. empty list, network-event mismatch despite a correct-looking page). The second is a *task-interpretation gap*: the verifier accepts an answer that

the page supports, but the answer is broader or narrower than what the task actually asks for (Appendix A.4). The supported claim is therefore narrower than "multi-agent solves skill induction": decomposition is *necessary* for net gains on a strong baseline, while evaluator alignment and verifier task grounding remain open problems we return to in Section 5.

4.5 Cross-Framework Transferability: A Case Study

Our induced skills are standalone Python modules calling the public `playwright.async_api` surface. They are not bound to OpenHands or to the multi-agent pipeline that produced them; any code-capable agent that can edit a file and import a Python module can use them. We test this directly: we drop the same skill files into the workspace of a generic Claude Code subagent (the `general-purpose` agent type, Sonnet backbone) and have it solve WebArena-Verified tasks under two conditions, both with the same minimal task prompt:

- **No skill**: the subagent receives only the task intent, the WebArena server URL, and login credentials, and must implement `final_automation.py` from scratch using only Playwright.
- **With skill**: the same prompt, but the workspace also contains the corresponding per-domain skill library produced by our multi-agent pipeline, and the prompt mentions that `from skill_library.<module> import ...` is permitted.

Both conditions are evaluated against the same WebArena server environment with the same evaluator configuration; everything else is identical.

Three positive cases across three domains (Table 5) share the same shape: the unaided subagent's failure is not a coding mistake but a knowledge gap about the underlying web application (Reddit runs on Postmill; Shopping and Shopping Admin run on the Magento storefront and admin panel respectively). On Reddit, it guesses `textarea[name="user[biography]"]` when the actual form field is `user_biography[biography]`. On Shopping Admin, it waits for the visible `<form>` on the bestsellers report and times out, because the bestsellers filter widget is a separately-injected hidden form. On Shopping, it scrapes a generic `<td>` sequence in the order-history table and inadvertently returns the order-id column when the task asks for the date. In all three cases, the failure is not a coding mistake but a missing piece of site-specific tribal knowledge that the induced skill encodes once; the subagent that imports the skill inherits the correct path for free.

With-skill scripts are shorter than the from-scratch versions (49 vs. 91 lines on Reddit, 50 vs. 115 on Shopping, 29 vs. 212 on Shopping Admin) and contain fewer defensive fallback chains: the no-skill scripts fall back through alternative selectors when the first guess fails (e.g., three candidate textareas on Reddit, two filter-form variants on Shopping Admin), while the skill versions delegate selector resolution to the imported function and proceed in a single attempt.

We frame this as a preliminary case study rather than a benchmark. Due to time and cost limitations, the three tasks are chosen to span three domains where matching skill functions exist in our induced library, and we leave a benchmark-scale evaluation to future work. What the study establishes is the lower bar for transfer: there is no adaptation step between agent frameworks, only a directory copy. Skills produced by one code agent are immediately usable by any other agent that can import a Python module.

5 Discussion

Prior skill-induction work [15, 16] establishes that self-generated skills can be a net positive on action-based agents. On the stronger code-agent substrate the picture changes: naive induction is not uniformly beneficial but can be actively harmful through abstraction drift and library pollution, while remaining harmless on domains where the gain is library-shaped rather than gate-shaped.

The skill-pipeline ablation sharpens this. Holding representation fixed (Playwright functions in every skill row), the difference between Naive skill, Self-verification, and Multi-agent is primarily explained by verification quality and its isolation mechanism. The pipeline shifts the macro average by +6.4 pp on the same code-native representation. The per-domain contrast is the cleanest evidence. On Reddit, where the gain is library-shaped (API-endpoint knowledge), all three gating regimes converge at the same +20.0 pp gain over no-skill, because any regime admits the underlying knowledge. On Shopping Admin, where tasks turn on derived computations vulnerable to confirmation bias, isolated verification recovers an 18.6 pp gap that prompt-level self-verification cannot close. The headline message is not that code agents make skill induction easier. A stronger substrate raises the bar that any skill-update policy must clear, and the binding constraint is verification quality rather than skill representation.

6 Limitations & Future Work

Sample-size limits on the head-to-head pipeline gains. All ablation comparisons are on a 104-task subset. We use this subset as a controlled analysis slice for like-for-like comparisons across tool settings and skill-pipeline variants, and rely on the per-domain patterns and case studies to explain where the gains come from. On this subset, the strongest statistical separations are between browser-enabled agents and Playwright-only scripting, and between our multi-agent pipeline and the action-based baselines. The smaller head-to-head margins between the top code-agent variants are better treated as directional on this sample; a larger benchmark-wide evaluation would still be needed to characterize how stable these exact margins are under different task samples.

Table 5. Cross-framework transferability case study. Same Claude Code subagent (Sonnet, `general-purpose` agent type), same task prompt, two conditions: with vs. without our induced skill library available for import. Three positive cases across three domains.

Domain	Task	No-skill	With-skill	Lines (no/with)	Failure mode w/o skill
Reddit	399 (change bio)	0.0	**1.0**	91 / 49	wrong textarea selector
Shopping Admin	1 (top brand Q1 2022)	0.0	**1.0**	212 / 29	filter form hidden, wait timeout
Shopping	117 (first purchase date)	0.0	**1.0**	115 / 50	scraped order-id column instead of date

Single backbone model. All configurations use Claude Sonnet 4 as the LLM backbone. The substrate-shift claim depends on the same model being usable through both action-based and code-native interfaces; we do not characterize how the gap behaves across model families or scales. We expect the qualitative direction (verification-gate matters more on stronger substrates) to generalize, but the absolute magnitudes will not.

Domain coverage and order effects. We exclude Map (continuous-valued outputs that the exact-match evaluator systematically rejects) and multi-domain tasks (out of scope for within-domain skill accumulation). Within each domain, skills accumulate in task-ID order across a single sequential pass; we do not run order-randomized replicates, so we cannot quantify how much of the observed gain depends on the particular induction trajectory. The pollution chains in Appendix A.1 suggest that order-sensitivity is real but bounded by the gate—a single false accept can poison the next two tasks but does not propagate indefinitely once the gate is in place.

Evaluator-side artifacts. WebArena-Verified's deterministic evaluator improves over the original benchmark but introduces its own boundary cases. Reddit network-event checks penalize UI clicks that succeed visually but record empty HARs, which mechanically inflates the gap between browser-only and hybrid configurations on Reddit (Table 3; Appendix A.3). The verifier-permissiveness rate (64.5% of failed tasks; Appendix B) is therefore best read as an upper bound on *semantic* false-accepts: some are evaluator-contract mismatches that no in-browser verifier can detect. Closing this gap requires either schema-aware verification or a benchmark with verifier-aligned answer contracts.

Future work. The residual failure pattern of our multi-agent pipeline (still `verify=accept` on 64.5% of failed tasks; see Appendix B) points to three directions. First, *typed answer contracts* that let the verifier mechanically check evaluator-shape requirements (e.g., `project id` vs. `project path`, `NOT_FOUND_ERROR` vs. empty list) rather than only page-level plausibility. Second, *layered skill promotion* that holds task-shaped abstractions to a higher admission bar than primitive site-navigation helpers. Third, a *benchmark-scale transferability evaluation* extending the case study in Section 4.5 to test whether directory-copy transfer holds across non-WebArena environments and across additional agent frameworks.

7 Conclusion

We presented a study of online skill induction on code-native web agents, where skills are standalone Playwright functions rather than compositions over a fixed action vocabulary. Three findings stand out. First, the substrate matters: code-native interaction yields substantially stronger no-skill baselines than action-based scaffolding. Second, on this stronger baseline, naive skill induction is no longer trivially helpful, shifting the bottleneck from skill representation to verification quality; our multi-agent pipeline addresses this by isolating solving from verification and gating updates. Third, skills in this form are structurally portable, transferring across agent frameworks as ordinary Python functions in our preliminary case study. Taken together, these results suggest a shift in focus: progress in online skill induction depends less on inventing new skill formats, and more on improving the substrate on which skills run and the verification that governs their admission. Because the representation is already executable code, portability follows naturally from it. Extending this pipeline to live websites beyond WebArena and testing transfer at scale under noisier evaluators are natural next steps.

References

[1] Browser Use Team. 2026. Browser Use. https://browser-use.com/. Framework for AI agents interacting with web browsers. Accessed: 2026-04-19.

[2] Weili Cao, Xunjian Yin, Bhuwan Dhingra, and Shuyan Zhou. 2026. Coding Agents are Effective Long-Context Processors. arXiv:2603.20432 [cs.CL] https://arxiv.org/abs/2603.20432

[3] Thibault Le Sellier De Chezelles, Maxime Gasse, Alexandre Drouin, Massimo Caccia, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Quentin Cappart, Graham Neubig, Ruslan Salakhutdinov, Nicolas Chapados, and Alexandre Lacoste. 2025. The BrowserGym Ecosystem for Web Agent Research. arXiv:2412.05467 [cs.LG] https://arxiv.org/abs/2412.05467

[4] 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 (2023), 28091–28114.

[5] Gonzalo Gonzalez-Pumariega, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. 2025. The unreasonable effectiveness of

scaling agents for computer use. *arXiv preprint arXiv:2510.02250* (2025).

[6] Amine El hattami, Megh Thakkar, Nicolas Chapados, and Christopher Pal. 2025. WebArena Verified: Reliable Evaluation for Web Agents. In *Workshop on Scaling Environments for Agents*. https://openreview.net/forum?id=94tlGxmqkN

[7] Yanna Jiang, Delong Li, Haiyu Deng, Baihe Ma, Xu Wang, Qin Wang, and Guangsheng Yu. 2026. SoK: Agentic Skills–Beyond Tool Use in LLM Agents. *arXiv preprint arXiv:2602.20867* (2026).

[8] Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. 2024. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 881–905.

[9] Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, et al. 2026. SkillsBench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670* (2026).

[10] Paloma Sodhi, SRK Branavan, Yoav Artzi, and Ryan McDonald. 2023. Step: Stacked llm policies for web actions. *arXiv preprint arXiv:2310.03720* (2023).

[11] Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, et al. 2025. Coact-1: Computer-using agents with coding as actions. *arXiv preprint arXiv:2508.03923* (2025).

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

[13] Aditya Bharat Soni, Boxuan Li, Xingyao Wang, Valerie Chen, and Graham Neubig. 2026. Coding agents with multimodal browsing are generalist problem solvers. In *Findings of the Association for Computational Linguistics: EACL 2026*. 6052–6069.

[14] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).

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

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

[17] Zora Zhiruo Wang, Sanidhya Vijayvargiya, Aspen Chen, Hanmo Zhang, Venu Arvind Arangarajan, Jett Chen, Valerie Chen, Diyi Yang, Daniel Fried, and Graham Neubig. 2026. How Well Does Agent Development Reflect Real-World Work? arXiv:2603.01203 [cs.AI] https://arxiv.org/abs/2603.01203

[18] Renjun Xu and Yang Yan. 2026. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *arXiv preprint arXiv:2602.12430* (2026).

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

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

[21] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854* (2023).

A Case Studies

This appendix collects the per-task evidence that supports the qualitative claims in Section 4.4. All case descriptions are reconstructed from the task-local artifacts (`agent_response.json`, `final_automation.py`, `eval_result_browsertool.json`, `verification_result.json`, `skill_update_result.json`) and the version-controlled skill-library commit log.

A.1 Skill Pollution Chains Under Naive Induction

Three concrete chains in which one false-accept update poisons multiple downstream tasks. Each chain is a single sequence inside one domain's skill repo, ordered by task-execution index.

A.1.1 Chain A: Shopping Admin 63 → 64 → 65. Task 63 fails with `eval_score=0`; the verify agent returns `accept` and the update commit adds two helpers to `ecommerce_admin.py`. The first helper (below, abridged) extracts the customer email by regex over the entire raw HTML of an order page:

```
async def get_customer_email_from_order(page, order_id):
    order_url = f"{base}/admin/sales/order/view/order_id
        /{order_id}/"
    await page.goto(order_url); await page.
        wait_for_timeout(3000)
    page_content = await page.content()
    email_match = re.search(
        r'[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}'
            ,
        page_content)
    return email_match.group() if email_match else None
```

The regex matches the *first* email anywhere on the page, which on Magento's order view is often the store-admin or shipping contact rather than the customer billing email. Task 64 imports this helper for a customer-lookup task and returns the wrong address. Task 65 imports both helpers, layers a customer-orders aggregation on top of the wrong customer mapping, and fails on every aggregated count.

A.1.2 Chain B: Shopping 146 → 147.

- Task 146 fails but the update commit adds an overfit `extract_product_dimensions_from_order` (centered on picture-frame products) to `ecommerce_orders.py`.
- Task 147 imports `find_dimensional_products_in_date_range` and `extract_product_dimensions_from_order`, applies the picture-frame parsing logic to a different product class, and fails.

A.1.3 Chain C: Shopping Admin 2 → 3 → 6. A cluster of best-seller parsing helpers (`parse_bestsellers_data`, `get_top_selling_products`, related product-type aggregation) is repeatedly written and rewritten across these tasks. Multiple wrong tasks were accepted and committed, which reinforced the wrong abstraction rather than correcting it. This is the qualitative pattern that motivates a layered-promotion design, which we leave to future work.

A.2 Confirmation-Bias Failures Under Self-Verification (Shopping Admin)

The five tasks below all regress from `pass` under no-skill to `fail` under self-verification. The first three are *script-shaped*: a derived computation disagrees with page evidence on the same screen and the verifier accepts the script anyway. The last two are *contract-shaped*: the failure is not in the script but in the choice of abstraction.

A.2.1 Task 15: "Get total reviews mentioning 'best' ". The agent imports `count_reviews_using_search_filter` from the skill library. The function attempts to type into the Magento admin's column-filter row using a positional CSS selector:

```
# Review column is "typically the 6th column"
filter_inputs = await page.locator(
    "table thead tr td input").all()
# Heuristic: the input at index 5 (or 4) is Review text
review_input = filter_inputs[5]
await review_input.fill(search_term)
await review_input.press("Enter")
count = await page.locator(
    "table tbody tr").count()  # also unfiltered
```

The selector matches inputs from *both* the column-filter row and the search-toolbar row, and indexing depends on whether the column visibility toggle hides any columns. On the All-Reviews page the index lands on the wrong column, so the filter does not narrow results to "best". The unfiltered tbody count returns **7**; the correct, filter-applied count is **2**. The no-skill baseline reads the same admin table without the skill and returns 2 correctly.

A.2.2 Task 77: "Get total Pending reviews". The imported `count_pending_reviews` pages through the admin Pending Reviews list with an over-broad row selector and a row-validity heuristic that does not actually filter to data rows:

```
while True:
    rows = await page.locator(
        "table tbody tr").all()       # matches header,
    if not rows:                      # filter, and data
        rows
        rows = await page.locator(
            "table tr").all()
    if not rows: break
    page_count = 0
    for row in rows:
        cells = await row.locator("td").all()
        if len(cells) < 3: continue
        review_id = (await cells[1]
            .inner_text()).strip()
        if not review_id or "ID" in review_id \
           or "Any" in review_id: continue
        try: int(review_id); page_count += 1
        except ValueError: continue
    total_count += page_count
    # ... pagination next-page click, no break on
```

```
    # already-counted rows when next-page click no-ops
```

The pagination loop never reliably terminates: when the "Next page" button is disabled but visible, the next iteration sees the same rows again and double-counts them. The function returns **10** for a single-page result. The pagination footer on the same screen reads "**5 records found**." The no-skill baseline (9 actions) reads that footer directly.

A.2.3 Task 78: "Get total Approved reviews". The agent imports only `login_to_magento_admin` and writes its own *efficient* counter that explicitly trades exactness for a hardcoded extrapolation:

```
# Sample first 10 pages
total_approved = 0; sample_pages = 10
for page_num in range(1, sample_pages + 1):
    rows = await page.locator(
        "table tbody tr").all()
    for row in rows:
        cells = await row.locator("td").all()
        if len(cells) > 3 and "Approved" in await
           cells[3].inner_text(): page_approved += 1
    total_approved += page_approved
    # ... click Next page or break
avg_per_page = total_approved / sample_pages
total_pages = 18  # observed manually
estimated_total = int(avg_per_page * total_pages)
return estimated_total  # 351, exact answer is 346
```

Both the page count (18) and the assumption that all pages have a stationary Approved fraction are observations from the agent's prior browsing session, not page reads at evaluation time. An exact existing skill (`count_approved_reviews`) and a direct pagination summary read (346) are both available; the verifier accepts the estimate anyway.

A.2.4 Task 3: "Get top-2 best-selling product names in 2022". The agent imports `get_top_selling_products` but *never awaits it*, falls back to a hardcoded observation from the skill-development browsing session (single-order rows on the bestsellers report), and returns `["Hera Pullover Hoodie","Sprite Yoga Strap 10 foot"]` based on highest single-order quantity (qty = 2). The correct answer requires aggregating all 2022 orders per product (Reports > Products > Bestsellers with date filter), which the no-skill baseline does. This is contract-shaped: the script ran, but it solved the wrong subproblem.

A.2.5 Task 62: "Customer email with the most orders in entire history". The skill library contains `get_customer_with_most_orders` and `get_customer_email_order_counts`—exactly the right functions. The agent spends 28 actions exploring the library, then writes a custom `Counter`-on-paginated-rows loop that picks up a customer with a high local count on the first page (`helloworld@yahoo.com`) instead of the global maximum (`janesmith456@yahoo.com`). The no-skill baseline (11 actions) sorts the admin Customers grid by order count and reads the top result.

A.3 Reddit: Skill-Helped Cases

Multi-agent improves over no-skill on `27, 28, 29, 66, 400, 401, 410` (7 tasks; 1 regression on `732`). The mechanisms below are reconstructed from case inspection across the broader skill-aware Reddit regime, including the closely related self-verification analysis. Tasks marked † are improvements under self-verification but not under multi-agent; we include them as additional evidence for the same mechanism, not as direct contributors to the +20.0 pp multi-agent gain.

A.3.1 Mechanism A: API endpoint knowledge for UI mutations. On UI-mutation tasks (upvote, comment, post submission) the no-skill agent reports `SUCCESS` but the network HAR is empty, because the browser-tool click did not trigger the expected POST and the network-event evaluator therefore fails. Skill code teaches the agent *how* the Postmill site's API works:

- `Task 410` (multi-agent improvement): "Reply to the first reply in this post with 'don't panic' " — POSTs to `/f/singularity/69404/-/comment/1042264` with `reply_to_comment_1042264[comment]=don't panic`.
- `Task 405`†: "Upvote the newest post in DIY forum" — skill function navigates to `/f/diy?sort=new`, finds the newest post, and POSTs to `/sv/<post_id>.json` with `choice=1`.
- `Task 625`†: "Create discussion post titled 'the effectiveness of deep learning' " — `create_discussion_post` POSTs to `/submit/deeplearning` with `submission[title]` and `submission[body]`.

This pattern is consistent with the multi-agent improvements on `400, 401` (also UI-mutation tasks), for which we did not collect detailed traces.

A.3.2 Mechanism B: Live re-navigation over hardcoded observations. On Books-forum extraction tasks the no-skill baseline hardcodes a stale post list collected during exploration and returns the wrong answer; skill-aware methods re-navigate live, often guided by skill-library docstrings that act as semantic specifications.

- `Task 66` (multi-agent improvement): "Among top 10 hottest Books posts, get titles from those recommending exactly one book." The no-skill agent hard-codes a post list and flags "Misty of Chincoteague" (a news article that mentions a book) as a recommendation. The skill agent re-navigates and returns the correct "I just finished reading The Hobbit…" post.
- `Task 67`†, `Task 69`†: same Books-forum pattern. On `69` ("URLs of organizations supporting local bookstores") the no-skill agent returns four URLs including `https://`-prefixed variants; the skill agent returns the single string `bookshop.org` as it appears in the post, matching the helper docstring ("URLs exactly as they appear").

The remaining multi-agent improvements (`27, 28, 29`) are workflow-edit tasks (bio edit, post edit, profile-style updates) that fall under the same Mechanism A category but were not deeply traced.

A.4 Contract-Drift Failures (Multi-Agent Residual Failures)

Cases where the page looks plausible to the verifier but the accepted answer is still wrong under either the benchmark contract or the task's intended semantic boundary. These are the dominant residual failures under the multi-agent pipeline.

A.4.1 GitLab Task 181: `NOT_FOUND_ERROR` vs. empty list. "Get whether my latest created issue with 'theme editor' in its title is closed." During skill development the agent observed zero results on `/dashboard/issues?. . . &search=theme+editor`, then hard-coded `return {"status":"NOT_FOUND_ERROR",. . . }` unconditionally in all three return paths of the function. At evaluation time the matching issue is present and not closed; the script returns `NOT_FOUND_ERROR` regardless. The no-skill baseline reads the issue list and returns `False`.

A.4.2 Shopping Task 144: false-positive food classifier. "How much I spent on food shopping Jan 15–31 2023, excluding shipping." Correct answer: $0.00 (no food orders in range). The agent uses skill helpers `get_all_paginated_orders` and `filter_orders_by_date_range` correctly, then writes a custom `is_food_item` classifier with ∼60 keywords (`organic`, `fresh`, `gourmet`, `nutrition`, `protein`, `supplement`). One non-food item matches and the script returns $82.26.

A.4.3 Shopping Task 159: "best storage option for 31 cards". The correct product holds 40 cards (smallest sufficient capacity); the agent hardcodes a navigation to a 160-card case (the largest option, prominently labeled "Nintendo Switch" in the link text). Even though a relevant skill module exists, the agent writes a pure-Playwright solution and selects on link text rather than reasoning about capacity. "Best" is interpreted as "largest" instead of "minimum sufficient."

A.4.4 GitLab Task 133: marginal context exhaustion. "How many commits did Eric make to a11yproject.com on March 2, 2023?" The agent spends all 78 turns manually browsing the GitLab commit history and never writes `final_automation.py`; the no-skill baseline writes the script in 76 actions. Action distribution: `BrowserGetState`×10, `BrowserScroll`×8, `BrowserClick`×5, `FileEditor`×7, zero touches on `final_automation.py`. This is best treated as marginal LLM-randomness inflated by skill-library exploration overhead.

Table 6. Verifier and update-gate behavior for the multi-agent pipeline on the 104-task subset. This table is restricted to the 99 tasks that emitted a complete verify/update artifact triple (`verification_result.json` and `skill_update_result.json` in addition to `eval_result`); the remaining 5 tasks are counted as completed in Table 7 but lack one of these gate artifacts and are excluded here. Two diagnostic rates: **20/31 = 64.5%** of failed tasks still received `verify=accept`, and **13/31 = 41.9%** of failed tasks still wrote skill updates back to the library.

Eval	Verify	Update	Count
pass	accept	true	42
pass	accept	false	23
pass	reject	false	3
fail	accept	true	13
fail	accept	false	7
fail	reject	false	11

B Verifier and Update-Gate Statistics

Table 6 reports the joint distribution of evaluator outcome, verify-agent verdict, and update-agent decision for the multi-agent pipeline on the 104-task subset. Five tasks are missing one or more pipeline artifacts (counted as failures in the main table) and are excluded here, leaving 99 tasks with full data.

The two diagnostic rows are `fail/accept/true` (false-accept polluting the library) and `fail/accept/false` (false-accept caught only by the update agent declining). Together they account for 20/31 of all failures with full data, which is the residual permissiveness referenced in Section 4.4.

C Claim-Aligned Significance Tests

Table 8 reports paired exact McNemar tests on the same 104-task subset used throughout the paper. We include only the comparisons directly tied to the paper's core claims: whether browser-enabled code agents outperform Playwright-only scripting, and whether the full multi-agent pipeline outperforms the action-based baselines. For completeness, we also include the two final-step comparisons whose direction is positive in the main tables but underpowered on this subset.

The appendix table is intended to bound, not inflate, the claims in the main text. The 104-task subset cleanly distinguishes large effects, such as the gap between browser-enabled variants and Playwright-only scripting, and the gap between the full multi-agent pipeline and the action-based baselines. The smaller final-stage gains (`Hybrid` over `Browser-only`, `Multi-agent` over `Hybrid`) remain positive but underpowered on this subset, so the main text treats them as best-average results rather than as statistically decisive separations.

Table 7. Skill-induction statistics on the 104-task subset for the multi-agent pipeline. "Completed" counts all 104 tasks that produced an `eval_result`; this is the full set, whereas Table 6 restricts to the 99 tasks that also emitted complete verify/update gate artifacts. "Candidate" means the solve agent proposed a skill update; "Accept" is the verify-agent verdict; "Applied" is whether the update agent wrote to the library; "Direct use" and "Rewrite" are task-level skill-adoption signals from `skill_adoption.jsonl`.

Domain	Tasks	Completed	Candidate	Accept	Applied	Direct use	Rewrite
GitLab	18	18	18	16	12	17	8
Reddit	30	30	29	24	14	21	8
Shopping	29	29	29	25	16	17	15
Shopping Admin	27	27	26	20	13	25	9
Total	104	104	102	85	55	80	40

Table 8. Claim-aligned paired exact McNemar tests on the 104-task subset. The "discordant" column reports wins unique to the left setting vs. wins unique to the right setting. Success totals are micro counts over the 104-task subset, whereas Table 2 reports macro averages over domains; the two aggregations therefore differ (e.g., Multi-agent's 68 here vs. 67.2% macro average there).

Comparison	Success totals	Discordant	p-value	Direction	Interpretation
Browser-only vs. Playwright-only	57 vs. 28	38 vs. 9	0.000025	Browser-only > Playwright-only	Strong support
Hybrid (both) vs. Playwright-only	60 vs. 28	40 vs. 8	0.000003	Hybrid > Playwright-only	Strong support
Multi-agent vs. BrowserGym	68 vs. 50	23 vs. 5	0.000912	Multi-agent > BrowserGym	Strong support
Multi-agent vs. ASI	68 vs. 56	17 vs. 5	0.016901	Multi-agent > ASI	Supported
Hybrid (both) vs. Browser-only	60 vs. 57	15 vs. 12	0.701108	Hybrid > Browser-only	Directional but underpowered
Multi-agent vs. Hybrid (both)	68 vs. 60	15 vs. 7	0.133801	Multi-agent > Hybrid	Directional but underpowered

D Skill Induction Statistics

Table 7 summarizes how often the multi-agent pipeline attempted to induce skills, how often the verifier accepted the task result, how often an update was actually written back to the shared library, and how often the solve trajectory directly reused or rewrote library functions. These numbers are computed on the same 104-task subset used throughout the paper.

Two patterns stand out. First, skill induction is attempted on nearly every task, so the pipeline is not selectively active only on a few hard examples. Second, update application is much more conservative than candidate generation, which is exactly the intended role of the verification and update gates. At the same time, the gap between `Accept` and `Applied` shows that a task can look locally plausible to the verifier without necessarily becoming reusable shared knowledge.

E Representative Successful Skills

The main text and earlier case studies necessarily focus on regressions, but the pipeline also induces genuinely useful reusable skills. Before the per-skill walkthroughs, Table 9 gives the top induced functions by import count across the multi-agent run, which makes it clear that the bulk of reuse is concentrated on small infrastructure primitives (login, navigation) rather than task-shaped helpers.

We then walk through four representative positive patterns—one from each domain—chosen to span the spectrum from heavily reused primitives to mid-complexity URL-aware helpers.

E.1 Primitive-Style Extraction Skill: `count_reviews_containing_term`

This helper from `magento_admin.py` is the cleanest positive example from Shopping Admin. It is induced during the review-counting tasks and then supports later tasks that require exact counts across paginated result tables. The full body is reproduced below (signature, docstring, and all logic):

```
async def count_reviews_containing_term(
        page: Page, search_term: str) -> int:
    """Count total reviews containing a term."""
    # Navigate to Marketing > All Reviews
    await page.click("a:has-text('Marketing')")
    await page.click("a[href*='review/product/index']")
```

Table 9. Top induced skill functions by import count across the 104-task multi-agent run. Counts are extracted from `import` statements in each task's `final_automation.py`. Logins and navigation primitives dominate; task-shaped helpers appear only at the long tail.

Function	Import count
login_to_magento_admin	19
gitlab_login	9
login_user (Reddit)	6
login_to_ecommerce_site	4
create_post (Reddit)	4
navigate_to_order_history	3
navigate_to_bestsellers_report_custom_range	3
count_commits_by_author_and_date	3
analyze_order_history	3
upvote_newest_post	2

```
    await page.wait_for_url("**/review/product/index/")

    total_count = 0
    while True:
        await page.wait_for_selector("table")
        review_cells = await page.locator("td").all()
        for cell in review_cells:
            text = await cell.inner_text()
            if search_term.lower() in text.lower():
                # filter to actual review cells:
                # review text is always >20 chars
                if len(text.strip()) > 20:
                    total_count += 1
        # Pagination: stop when Next is disabled
        next_btn = page.locator(
            "button:has-text('Next page')")
        if await next_btn.is_disabled(): break
        await next_btn.click()
        await page.wait_for_timeout(1000)
    return total_count
```

Three properties make this skill survive reuse where the failures in Appendix A.2 did not. (i) The pagination loop terminates on a directly observable page affordance (`is_disabled()` on the *Next page* button) rather than a hardcoded page count or sample average. (ii) The substring match is gated by an explicit cell-length filter, which avoids matching short header or filter rows that triggered double-counting in `count_pending_reviews`. (iii) The function exposes `search_term` as an argument, so the skill is reused across multiple tasks (e.g. `Task 11` "disappointed" returns the correct **6** where the no-skill agent reads only the first page and returns 2; `Task 13` "decent" returns **2** where a broader unscoped match returns 7) instead of being overfit to the inducing example.

E.2 Workflow/API Skill: Reddit Mutation Endpoints

The most useful Reddit skills do not primarily encode semantic reasoning; they encode how the site's mutation endpoints actually work, in a form that the network-event evaluator can verify. A representative pair from `reddit_navigation.py`:

```
async def reply_to_comment(page: Page,
        target_username: str,
        reply_message: str,
        comment_id: str = None) -> dict:
    """Reply to a specific user's comment."""
    user_link = await page.query_selector(
        f'a[href="/user/{target_username}"]')
    if not user_link: return {"success": False, ...}
    # Walk up to the comment container, find its
    # Reply link (scoped, not page-global).
    container = await user_link.evaluate_handle(
        'el => el.closest("article") '
        '|| el.closest("li")')
    reply_link = await container.query_selector(
        'a:has-text("Reply")')
    await reply_link.click()
    # The Reply link navigates to a comment-form
    # page that POSTs to /f/<forum>/<post>/-/comment/
    # <parent_id>, which is what the network-event
    # evaluator actually checks.
    textarea = await page.query_selector('textarea')
    await textarea.fill(reply_message)
    submit = await page.query_selector(
        'button:has-text("Post")')
    await submit.click()
    return {"success": True}

async def upvote_post(page, post_element) -> dict:
    """Upvote a specific post element via its
    own button container, not a page-global query."""
    upvote_button = await post_element.query_selector(
        'button[title*="upvote"], '
        'button[aria-label*="upvote"]')
    await upvote_button.click()  # POSTs to
    # /sv/<post_id>.json with choice=1
    return {"success": True}
```

Both helpers share a structural property absent from the failure cases: every selector is scoped to the relevant comment or post container (`closest("article")`, `post_element.query_selector(...)`) rather than executed against the whole page. This is what lets the same skill be reused across different forums and post targets without rebinding to a specific URL or hardcoded element ID. The corresponding HAR cleanly shows the expected POST (e.g. to `/f/singularity/69404/-/comment/1042264` for the reply, or `/sv/<post_id>.json` for the upvote), which is precisely what the network-event evaluator checks; the no-skill agent often dispatches a UI click that succeeds visually but records an empty HAR. The skill is useful not because it replaces task-specific judgment but because it encodes the procedural contract between the visible UI and the evaluator.

E.3 URL-Parameter Skill: `get_personal_projects_with_star_count` (GitLab)

The single GitLab improvement under skills (`Task 172`: "project IDs of my personal projects that received no stars") comes from a small skill that exploits two URL-level affordances rather than scraping the rendered page:

```
async def get_personal_projects_with_star_count(
        page, base_url, username, star_count=0):
    """Get personal projects with given star count."""
    # Affordance 1: list filter via URL parameter
```

```
    url = f"{base_url.rstrip('/')}/" \
          f"?personal=true&sort=name_asc"
    await page.goto(url)
    await page.wait_for_load_state('networkidle')

    # Affordance 2: every project's star count is
    # the inner text of its /-/starrers link
    star_links = await page.query_selector_all(
        'a[href*="/-/starrers"]')
    matching = []
    for link in star_links:
        href = await link.get_attribute('href')
        text = (await link.inner_text()).strip()
        if text == str(star_count) \
           and f'/{username}/' in href:
            project = href.split(f'/{username}/')[-1] \
                .replace('/-/starrers', '')
            matching.append({'name': project, ...})
    return {"status": "SUCCESS",
            "projects": matching}
```

This contrasts directly with the GitLab failure cases in Appendix A.4: `Task 181` hardcoded a `NOT_FOUND_ERROR` return, and `Task 133` ran out of turns scrolling commit history. Here the agent uses GitLab's `?personal=true` URL parameter (a primary-source filter) instead of post-filtering all projects, and reads the star count directly from the `/-/starrers` link rather than parsing free-text labels. The no-skill baseline (84 actions) navigates the user profile and collects every project ID it sees, including group projects and projects with non-zero stars, returning 7 wrong IDs; this skill returns the correct 6 (`181, 184, 188, 189, 190, 193`) by aligning with the contract the GitLab UI itself uses.

E.4 Heavy-Reuse Login Primitive: `login_to_magento_admin`

The most-imported induced function across the multi-agent run (Table 9) is not a complex extractor. It is a 12-line login helper:

```
async def login_to_magento_admin(
        page, base_url,
        username="admin",
        password="admin1234") -> bool:
    """Login to the Magento admin panel."""
    await page.goto(f"{base_url}/admin")
    await page.fill(
        "input[placeholder='user name']", username)
    await page.fill(
        "input[placeholder='password']", password)
    await page.click("button:has-text('Sign in')")
    await page.wait_for_url(
        "**/admin/admin/dashboard/")
    return True
```

The function is imported **19 times** across the Shopping Admin tasks; each importer saves the 4–5 actions it would otherwise spend exploring the login form, and—more importantly—never has to re-discover the redirect target (`/admin/admin/dashboard/`) that signals success. Three sibling primitives in Table 9 (`gitlab_login`, `login_user`, `login_to_ecommerce_site`) play the same role for the other domains, with import counts of 9, 6, and 4 respectively. None of these is impressive in isolation; together they illustrate why heavy-reuse rather than per-task novelty is the metric that matters for a skill library: the four login helpers alone account for *38 of the 71 imports* we recorded across the run, while the task-shaped helpers in Table 9 (excluding the login and navigation primitives) account for only 12. This distribution also explains why our gating mechanism is asymmetric: a false-accept on a primitive like `login_to_magento_admin` would propagate into 19 downstream tasks at once, which is exactly the worst-case the verify and update agents are designed to block.

F Prompt Appendix

This section lists the exact prompts used for the six experimental settings compared in the paper. All prompts are copied into this paper repository so the appendix is self-contained. The tool-only and naive-skill prompts are extracted from `webarena_test/example_verified.py`; the self-verification and multi-agent prompts are copied verbatim from the corresponding `webarena_test/skill_prompt/*` source files.

F.1 Tool-Only Prompts

These three prompts are used by the no-skill code-agent rows and define the tool ablation in Table 3. They contain only the tool-use instructions; no skill library, no verification, and no separate verify or update stage are involved.

F.1.1 Browser only (the `Browser-only` row). Solve agent has access to the browser tool only; the terminal is disabled. Used to produce the `Browser-only` row of Table 3.

```
## Tools Available
1. Browser tool for web navigation (use browser_* actions
   )
2. Terminal for command execution (non-browser tasks only
   )

IMPORTANT: You must use the browser tool for all web
   interactions. Do NOT write
Playwright scripts or automation code. Interact with the
   browser directly using
the browser tool actions.

## Workflow
1. Use the browser tool to navigate to the website
2. Complete the task using browser tool actions
3. Extract/verify the required information
4. Save your final response to ./agent_response.json
```

F.1.2 Playwright only (the `Playwright-only` row). Solve agent has access to the terminal only; the browser tool is disabled, so all interaction goes through Playwright scripts executed in the shell. Used to produce the `Playwright-only` row of Table 3.

```
## Tools Available
1. Python + Playwright (already installed) for automation
   . Save your code in
   ./funcs/
2. Terminal for command execution
```

Table 10. Prompt structure by experimental setting.

Setting	Browser	PW	Skill Lib	Self-Verify	Verify Agent	Update Agent
Browser only	yes	no	no	no	no	no
Playwright only	no	yes	no	no	no	no
Both	yes	yes	no	no	no	no
Naive skill	yes	yes	yes	no	no	no
Self-verification	yes	yes	yes	yes	no	no
Multi-agent	yes	yes	yes	no	yes	yes

```
IMPORTANT: You must use Playwright scripts for all web
     interactions. Write
Python automation code using Playwright and run it via
     terminal. Do NOT use the
browser tool directly.

## Workflow
1. Write a Playwright script to navigate to the website
     and complete the task
2. Save your automation script in ./funcs/
     final_automation.py. Refer to
   ./template.py for a template.
3. Run and test your script via terminal
4. Extract/verify the required information
5. Save your final response to ./agent_response.json
```

F.1.3 Both (the `Hybrid (both)` row, also `Code agent (no skill)`).

Solve agent has simultaneous access to the browser tool and the terminal but no skill library. Used both as the `Hybrid (both)` row of Table 3 and as the `Code agent (no skill)` row of Tables 2 and 4.

```
## Tools Available
1. Browser tool for web navigation
2. Python + Playwright (already installed) for automation
     . Save your code in
   ./funcs/
3. Terminal for command execution

You can choose to use either the browser tool or
     Playwright scripts for web
interactions.

## Workflow
1. Navigate to the website and complete the task
2. Extract/verify the required information
3. Save your final response to ./agent_response.json
4. Save your final Playwright automation script in
   ./funcs/final_automation.py. Refer to ./template.py
        for a template. Write
   test code to check the final_automation.py script is
        correct.
```

F.2 Skill-Pipeline Variants

The three rows of Table 4 (`Naive skill`, `Self-verification`, `Multi-agent`) all share the hybrid Browser+Playwright tool stack of the `Both` setting and add the skill library, but differ in how the skill update is gated. The prompts below are organized by pipeline stage (solve / verify / update); the same `Multi-agent` row uses three prompts in sequence.

Solve-stage prompts. The solve agent always has skill-library read access. The three prompts below replace each other one-for-one across the three skill-pipeline rows.

F.2.1 Naive skill (solve prompt for the `Naive skill` row).

The solve agent reads, imports, and writes back to the skill library in a single conversation, with no separate verification step. This corresponds to the `Naive skill` row of Table 4.

```
## Workflow
1. Import existing functions from ./skill_library
     whenever possible.
2. Navigate to the website and complete the task.
3. Extract/verify the required information.
4. Save your final response to ./agent_response.json.
5. Save your final Playwright automation script in
   ./funcs/final_automation.py. Refer to ./template.py
        for a template. Write
   test code to check the final_automation.py script is
        correct.
6. After verification, extract any new reusable functions
      into ./skill_library
   for future use.
```

F.2.2 Self-verification (solve prompt for the `Self-verification` row).

The same single-conversation solve agent, with additional in-prompt instructions to cross-check the script output against page evidence and prefer the page when they conflict. The `Self-Verify` column of Table 10 is on for this row only. No separate verify or update agent is invoked.

```
### Step 1: Review Skill Library & Semantic Check
- Read `./skill_library/` to understand what functions
     are available.
- Identify which functions are relevant to the current
     task.
- **Avoid task-specific functions** (e.g., `
     find_wireless_earphones_price_range`). Prefer
     generic parameterized functions (e.g., `
     find_product_price_range(search_term)`). If only
     task-specific functions exist, adapt or rewrite them
     .
- **Critical semantic check before using any function**:
     Read the function's code and verify that its logic
     matches what the task actually asks for. For example
     :
  - If the task asks for "issues I created", the function
       must filter by `author`, NOT by `assignee`.
  - If the task asks for "reviews containing X", the
     function must search the full review text, not
     just truncated previews.
```

```
  - If the function's filter/query semantics don't match
      the task's requirements, **do NOT use it** - write
       a corrected version.

### Step 2: Execute the Task
- {tools_step}
- Use skill library functions to complete the task.
- Collect the results.

### Step 3: Verify ALL Results on Page (MANDATORY -
     STRICT RULES)
After obtaining results, you MUST verify every data point
      against the actual
page. This is not optional and the rules below are
     absolute.

**Verification procedure:**
- Navigate to the relevant page yourself using the
     browser.
- Use a **different method** than your script used. For
     example, if the script
  scraped table rows, use the page's built-in search/
       filter instead. The goal
  is independent evidence, not re-execution.
- For counts: find the page's own count indicator (e.g.,
     "346 records found")
  or manually count items if the number is small.
- For specific values: check the page shows the same
     value.
- For boolean results: navigate to the item and confirm
     its status.
- For lists: spot-check at least 2-3 items from the
     result.
- If the result is zero (0): confirm by checking the page
      shows no matches.

**Mandatory conflict rules (absolute - no exceptions):**

1. **Page truth wins.** If the page shows a different
     answer than your script
   produced, the PAGE IS CORRECT. Do not trust the script
        over the page.

2. **Exact answers required.** When the task asks for a
     count, number, name,
   or specific value, the answer must be exactly correct.
        "351 vs 346" means
   your script is wrong. "10 vs 5" means your script is
        wrong. There is no
   acceptable margin of error for exact-answer tasks.

3. **Do not rationalize discrepancies.** If the page
     shows 5 items but your
   script says 10, do NOT explain it away as "hidden DOM
        elements" or
   "additional data not rendered." If you cannot see it
        on the page, it does
   not count.

4. **Do not use sampling or estimation** when the page
     provides an exact count.
   If the admin panel says "346 records found", that is
        the answer - do not
   sample pages and extrapolate.

**If verification reveals a mismatch:**
1. The page answer is correct. Your script has a bug.
2. Fix the script: correct pagination, filtering,
     counting, or selector logic.
3. Re-run and re-verify until the script matches the page
      exactly.
4. If you cannot fix the script, use the page answer
     directly.

### Step 4: Save Results
- Save your final response to `./agent_response.json`.
- Save your final Playwright automation script in `./
     funcs/final_automation.py`. Refer to `./template.py`
      for a template. Write test code to check the `
     final_automation.py` script is correct.

**IMPORTANT: Do NOT modify `./skill_library/`.** The
     skill library is read-only
during the solve phase. A separate update agent will
     handle skill updates after
verification.
```

F.2.3 Multi-agent solve (solve prompt for the `Multi-agent` row).

The solve agent submits its answer and a candidate skill update but *cannot* write to the skill library directly; the update is gated on the verify and update agents below. This is the first of the three multi-agent stages.

```
### Step 1: Review Skill Library & Semantic Check
- Read `./skill_library/` to understand what functions
     are available.
- Identify which functions are relevant to the current
     task.
- **Avoid task-specific functions** (e.g., `
     find_wireless_earphones_price_range`). Prefer
     generic parameterized functions (e.g., `
     find_product_price_range(search_term)`). If only
     task-specific functions exist, adapt or rewrite them
     .
- **Critical semantic check before using any function**:
     Read the function's code and verify that its logic
     matches what the task actually asks for. For example
     :
  - If the task asks for "issues I created", the function
       must filter by `author`, NOT by `assignee`.
  - If the task asks for "reviews containing X", the
       function must search the full review text, not
       just truncated previews.
  - If the function's filter/query semantics don't match
       the task's requirements, **do NOT use it** - write
       a corrected version.

### Step 2: Execute the Task
- {tools_step}
- Use skill library functions to complete the task.
- Collect the results.

**Answer quality reminders:**
- **Exact answers required.** When the task asks for a
     count, number, name,
  or specific value, the answer must be exactly correct.
       There is no
  acceptable margin of error for exact-answer tasks.
- **Do not use sampling or estimation** when the page
     provides an exact count
  or value. If the admin panel says "346 records found",
       that is the answer.
- **Page over script.** If your automation script
     produces a different result
  than what the page shows, trust the page - the script
       may have bugs in
  pagination, filtering, or selector logic.

### Step 3: Save Results
- Save your final response to `./agent_response.json`.
```

```
- Save your final Playwright automation script in `./
    funcs/final_automation.py`. Refer to `./template.py`
     for a template. Write test code to check the `
    final_automation.py` script is correct.

**IMPORTANT: Do NOT modify `./skill_library/`.** The
    skill library is read-only
during the solve phase. A separate update agent will
    handle skill updates after
verification.
```

Verify- and update-stage prompts. These two prompts are only invoked in the `Multi-agent` row. Each runs in an isolated conversation that has no access to the solve trajectory.

F.2.4 Multi-agent verify (`Multi-agent` row, stage 2).

Independent agent with its own browser context. Cross-checks the solve agent's submitted answer against live page evidence and runs `final_automation.py` once. Outputs `verification_result.json` with verdict $\in$ {`accept`, `reject`, `unclear`}.

```
You are the verification agent for a completed web task
    run.

Your job is only to verify whether the submitted final
    result for the current
task is correct.

## Task
**Objective:** {task}

**Starting URL(s):** {urls_text}
{login_info}

## Available Workspace Files

- `./agent_response.json`
- `./funcs/final_automation.py` (if present)
- `./example_logs.jsonl`

## Verification Goal

Determine whether the submitted final answer is actually
    correct for this task.
You are verifying the answer, not auditing the solve
    process and not proposing a
better solution.

You MUST do BOTH of the following - not just one:
- run `./funcs/final_automation.py` (if present) and
    record its output
- use the browser UI to independently verify the same
    data point (e.g., use
  the site's built-in search, filters, or page counts)

Then cross-check the two results. If they agree, you have
     strong evidence.
If they disagree, the **browser/page result wins** - the
    script may have bugs.

**Script failures**: Do NOT debug, fix, or retry the
    script. Run it once.
If it fails, errors out, or hangs - that counts as
    negative evidence against
the submitted answer. Move on to browser verification
    immediately.

You may also:
- read saved files to understand the submitted answer
- write and run your own verification scripts
- use the task statement as the source of truth

You MUST NOT:
- accept a result based on script output alone without
    browser verification
- accept a result based on browser alone without at least
     attempting to run
  the script (unless no script exists)
- debug, fix, or re-run the script - run it exactly once
    as-is
- modify the skill library
- propose skill updates
- reinterpret the task into a narrower or different
    question just to justify the
  submitted answer
- spend time exploring alternative solutions once you
    have enough evidence to
  accept, reject, or mark unclear

## Required Procedure

1. Read `./agent_response.json` to identify the submitted
     answer and its
   `task_type` (RETRIEVE, MUTATE, or NAVIGATE).
2. **Follow the type-specific verification below.**
3. Read `./funcs/final_automation.py` to understand the
    script's approach.
4. **Run the script once** and record its output.
5. **Cross-check**: compare your browser evidence vs
    script output vs
   submitted answer. If they all agree -> accept. If
       browser and script
   disagree -> browser wins. If browser disagrees with
       the submitted
   answer -> reject.

## Type-Specific Verification (Step 2)

Adapt your browser verification based on the task type:

### RETRIEVE tasks
- Independently look up the same data on the page using
    the site's built-in
  search, filters, or page counts.
- Compare your independently obtained answer with the
    submitted
  `retrieved_data`.
- If the values differ (even slightly), the page is
    correct.

### MUTATE tasks
- **Do NOT re-perform the action.** Do not click upvote,
    submit forms, post
  comments, or trigger any state change yourself.
- Instead, check whether the mutation's **effect is
    already visible** on the
  page:
  - For upvotes: check the post's current vote count or
      voted state indicator.
  - For comments/posts: check if the new comment or post
      appears on the page.
  - For edits: check if the edited content reflects the
      expected changes.
  - For deletions: check if the item is no longer visible
      .
- If the expected effect is NOT visible, reject - the
    mutation did not succeed.
- If the expected effect IS visible, accept.
```

```
### NAVIGATE tasks
- Check whether the **final page URL and visible content
    ** match the task
  objective.
- Run the script and verify it lands on the correct page
    with the expected
  elements visible (e.g., correct heading, correct filter
      state, correct
  project/issue/product displayed).
- If the script navigates to a wrong page or the final
    URL does not match
  the intent, reject.

## Mandatory Conflict Rules

These rules are absolute. You MUST follow them even if
    the difference seems
small or explainable.

- **Page truth wins.** If your independent page evidence
    gives a different
  answer than the submitted answer, you MUST choose `
      reject`. No exceptions.
  Do NOT rationalize the difference as "close enough", "
      within margin",
  "sampling error", or "DOM contains hidden data". A
      wrong number is wrong
  even if it is close.

- **Exact answers required.** When the task asks for a
    count, number, name,
  or specific value, the answer must be exactly correct.
      "351 vs 346" is a
  rejection, not an acceptable approximation. "10 vs 5"
      is a rejection.

- **Do not trust the script over the page.** If `
    final_automation.py` returns
  one answer but the page UI shows a different answer,
      choose the page. The
  script may have bugs in pagination, filtering, counting
      , or selector logic.
  The page is the source of truth.

- **Do not invent explanations for discrepancies.** If
    the page shows 5 items
  but the script says 10, do NOT speculate that "the DOM
      has hidden items" or
  "additional data not visually rendered." If you cannot
      see it on the page,
  it does not count.

If the evidence is insufficient or the task wording is
    genuinely ambiguous,
choose `unclear`.

## Output

Write `./verification_result.json` as:

```json
{{
  "verdict": "accept|reject|unclear",
  "reason": "short explanation",
  "verified_answer": "<your own independently verified
      answer or null>",
  "page_evidence": "<what you observed on the page>",
  "script_evidence": "<what the script returned, if run
      >",
  "conflict": true,
  "used_final_automation": true,
  "page_check_performed": true
}}
```

Set `"conflict": true` if your page evidence differs from
     the submitted answer
in any way. Set `"conflict": false` if they agree exactly
    .

You must write `./verification_result.json` before
    finishing.
```

F.2.5 Multi-agent update (`Multi-agent` row, stage 3).

Invoked only when verify returns accept and the candidate update is non-empty. Operates on a clean copy of the skill library restored from version control and applies targeted edits, producing `skill_update_result.json`. Has no browser access.

```
You are the skill update agent.

Your job is to update `./skill_library/` only if the
    previous solve run both:

1. proposed a skill update in `./{
    candidate_skill_update_filename}`
2. passed verification in `./{
    verification_result_filename}`

## Available Workspace Files

- `./skill_library/`
- `./skill_adoption.jsonl`
- `./{candidate_skill_update_filename}`
- `./{verification_result_filename}`
- `./funcs/final_automation.py`
- `./agent_response.json`

## Rules

- Only update the skill library if the proposal and
    verification together justify it.
- Prefer updating existing reusable skills over writing
    task-specific helpers.
- If no update should be applied, leave `./skill_library
    /` unchanged.

## Output

Write `./{skill_update_result_filename}` as:

```json
{{
  "applied_update": true,
  "updated_files": ["..."],
  "reason": "short explanation"
}}
```
```