FROM PAGES TO TOOLS: GENERATING AND INTEGRATING PAGE-COUPLED TOOLS FOR WEB AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Structured tools offer web agents an alternative to repeated low-level browser interactions, but exposing tools alone does not guarantee effective use. The design problem spans both tool construction and agent integration: which application capabilities should be exposed, when they should be available, and how their descriptions and execution feedback should enter the agent loop. We introduce **PACT** (**PA**ge-**C**oupled **T**ools), a framework for generating and integrating page-coupled tools. PACT derives reusable, single-function capabilities from page and application evidence, independently verifies their semantics, scope, execution, and agent-facing selection, and dynamically registers tools according to the live page state. Tool execution is bounded from higher-level agent reasoning, leaving tool selection, composition, and subsequent decisions to the calling agent. On 812 WebArena tasks, PACT improves success across all 16 agent–model configurations and reduces token usage in most of them. Tool-set and integration studies further show that tool quality and information exposure materially affect both performance and resource use. Additional evaluations across model sizes, web-agent fine-tuned models, WebArena Verified, and our mock websites further support the generality of PACT while revealing model- and configuration-dependent tradeoffs.

1 INTRODUCTION

Web agents pursue natural-language goals by interacting with web interfaces in a human-like manner (Zhou et al., 2024; Deng et al., 2023; He et al., 2024). They typically follow sequential state–reasoning–action steps (Yao et al., 2022), repeatedly reasoning over the current page and issuing low-level actions such as clicking, typing, scrolling, and navigation (Deng et al., 2023; Zhou et al., 2024; Lù & Reddy, 2026). This repeated control-locating and state reconstruction can introduce substantial redundancies. To improve efficiency, recent works mine reusable skills that encapsulate common action sequences across tasks (Wang et al., 2025; 2024). However, because these skills are often derived from successful trajectories, they can remain overspecified around concrete UI action sequences, still requiring substantial perception and making them sensitive to visual evidence.

WALT (Prabhu et al., 2026) addresses this limitation by moving beyond skills defined as low-level UI action sequences. Instead, it reverse-engineers website-provided functionality, such as search, filtering, and sorting, into reusable high-level tools (Lou et al., 2026). WALT follows a demonstrate–generate–validate pipeline. An agent first explores the website and demonstrates candidate functionality to collect execution traces. A separate agent then converts these traces into executable tools with structured input schemas, which are further optimized and validated against test inputs.

Despite the advance of structured tools from WALT, several limitations remain: **(L1)** its automated tool-discovery pipeline depends on browser-agent *exploration and demonstration trajectories*, making tool coverage dependent on what the *generating policy* discovers and demonstrates; **(L2)** its tools can combine deterministic operations with navigation, UI interaction, extraction, and agentic steps, mixing tool execution with model-driven reasoning; **(L3)** all the tools are largely exposed to the agent up front, regardless of their relevance to the current interaction state. As the tool set grows, this can introduce unnecessary context and tool-selection overhead, motivating selective tool exposure according to the live page state; and **(L4)** learned tools are largely integrated as additional actions in the agent's existing action space, leaving the broader tool–agent integration policy, including tool exposure, page observations, and execution feedback, largely unexplored.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

These limitations become particularly salient with the emerging Web Model Context Protocol (WebMCP[1]), which enables web applications to expose structured functionality directly to agents. Unlike static externally constructed tools, WebMCP couples tools to the active document and supports dynamically registering or removing them as the web page state changes. This design direction motivates reusable web capabilities that are grounded in application functionality, coupled to the live UI, and exposed only when applicable, while leaving higher-level planning to the calling agent.

We address these limitations with a framework for generating and integrating **PA**ge-**C**oupled **T**ools (**PACT**). Rather than constructing tools from successful agent trajectories, our method derives reusable, single-function capabilities directly from pages and applications, and independently verifies their functionality. Tool execution is kept bounded and separate from the agent's own reasoning, while tool availability is dynamically tied to the live page state, allowing tools to be registered or removed as their applicability changes. Beyond tool generation, we treat tool–agent integration as a first-class design problem, allowing the agent to combine tools with the agent's predefined actions while controlling access to page observations, tool descriptions, and execution feedback. To evaluate whether these tools provide benefits beyond a particular agent or model, we test them across multiple combinations of agent frameworks and backend models. Fig. 3 shows that our tools improve success rates and generally reduce token usage. We further vary the information exposed through the tool–agent interface and compare alternative tool sets and optimization strategies, allowing us to study not only whether page-coupled tools are effective, but also how their construction and integration affect downstream agent performance. Our main contributions are:

- We introduce **PACT**, a framework for generating *page-coupled tools* from page and application evidence. PACT derives reusable single-function capabilities, independently verifies them, keeps execution bounded from higher-level agent reasoning, and dynamically registers tools according to the live page state.
- We treat tool–agent integration as a first-class design problem. PACT allows agents to combine page-coupled tools with predefined actions while controlling the exposure of page observations, tool descriptions, and execution feedback throughout the interaction loop.
- We conduct a systematic evaluation on six WebArena applications across four agent frameworks and four backend models, on 812 tasks. We further vary tool sets, information exposure, and tool optimization to study when PACT improves performance and how its benefits depend on model and integration choices.

2 Related Work

Web-agent designs. Web agents improve observation, planning, and control through abstraction, search, memory, and specialized architectures (Yang et al., 2025; Dihan et al., 2025; Wang et al., 2026; Marreed et al., 2025). Complementary approaches use policy learning, synthetic trajectories, and learned simulations (Guo et al., 2026; Lù & Reddy, 2026; Xiao et al., 2026). PACT complements these agent-side designs through application-facing tool generation and agent integration.

Reusable web skills and tools. Reusable web capabilities include experience-induced workflows (Wang et al., 2024), verified programmatic skills (Wang et al., 2025), and parameterized browser programs distilled from completed tasks (Lu et al., 2026). WALT synthesizes and validates reusable tools for website functionality (Prabhu et al., 2026). PACT shares this focus, emphasizing generation from page and application evidence, independent verification, and page-state-dependent tool availability.

Tool interfaces and agent integration. MCP provides client–server interfaces for tools and context (Anthropic, 2024), while WebMCP proposes document-level tool registration and invocation (Web Machine Learning Community Group, 2026). WebMCP's developer guidance addresses single-function tools, registration scope, and observable execution outcomes (Klepper, 2026a). At the service level, ToolPro represents multi-step interactions as executable programs with control flow and effect-aware replay (Liu et al., 2026). Lou et al. (2026) report model-dependent gains and token overhead, and advocate separating agentic reasoning from tool execution. PACT adopts this separation through bounded tools. Our integration studies fix the tool set and vary page-observation and tool-definition exposure while retaining predefined actions and incorporating execution feedback.

[1] https://webmachinelearning.github.io/webmcp/

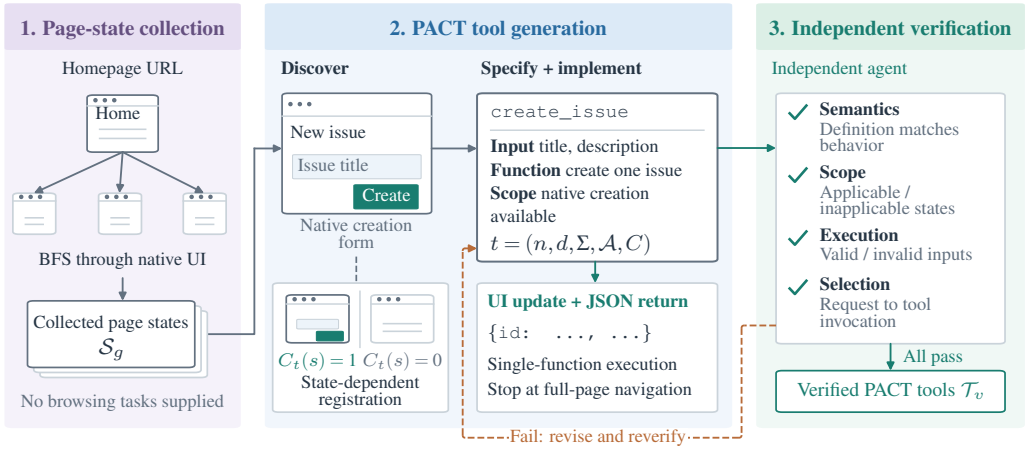

Figure 1: **PACT tool-generation methodology.** Only candidate tools passing all verification checks enter the verified set $\mathcal{T}_v$. Any verification failure triggers tool revision and subsequent reverification.

Evaluation of interactive agents. Existing benchmarks cover visually grounded web tasks (Koh et al., 2024), diverse visual-interaction environments (Liu et al., 2025), difficult information retrieval (Wei et al., 2025), and long-horizon browsing (Jang et al., 2026). OSWorld-MCP evaluates tool invocation alongside desktop GUI interaction (Jia et al., 2026), while other work examines limitations of existing web-agent evaluations (Xue et al., 2025). Our paired evaluations on WebArena (Zhou et al., 2024), supplemented by WebArena Verified (hattami et al., 2025), assess how tool generation and integration affect task success and resource use.

3 Methodology

3.1 Formal Definition of Page-Coupled Tool

We follow WebMCP protocols (Klepper, 2026a) to define our tools. Each tool $t \in \mathcal{T}$ is specified by its agent-facing interface and registration scope. More formally, we define a tool as a tuple:

$$t = (n, d, \Sigma, \mathcal{A}, C), \tag{1}$$

where n is its name, d its description of its intended behavior, Σ its input schema, $\mathcal{A}$ a set of annotation flags indicating whether a tool is read-only or returns untrusted content that should not be treated as instructions, and $C : \mathcal{S} \to \{0, 1\}$ its registration condition over the page-state space $\mathcal{S}$.

A page state $s \in \mathcal{S}$ comprises the application data, DOM structure, and UI conditions that determine the current page behavior. For each tool t, C_t denotes its registration condition, where $C_t(s) = 1$ if and only if the required elements are present and the tool is applicable in state s. The browser therefore dynamically updates the registered tool set as the page state changes:

$$\mathcal{R}(s) = \{t \in \mathcal{T} \mid C_t(s) = 1\}. \tag{2}$$

We then represent an execution of a registered tool t as:

$$(s, x) \xrightarrow{t} (s', r), \tag{3}$$

where x is the tool input, s' the resulting page state, and r the structured return. Together, (s', r) provides complementary feedback: s' captures the observable outcome, while r reports execution details such as success, failure, input validity, or extracted information.

3.2 PACT Tool Generation

For a given website, our generation method takes the URL of the website homepage as input and outputs a set of verified tools $\mathcal{T}_v$ for the website. As shown in Figure 1, our generation method **(1)** constructs a set of targeted page states $\mathcal{S}_g \subseteq \mathcal{S}$, **(2)** generates a set of candidate tools, and **(3)** verifies the candidate tools against protocols where failed tools are sent back for revision and reverification.

3.2.1 PAGE STATES COLLECTION

We develop an agent skill to collect a target set of page states $\mathcal{S}_g$ for tool generation (Appendix A.1.1). Starting from the website home page, a coding agent performs breadth-first exploration using only page-native interactions to discover reachable states. Importantly, this process is bounded by the application itself, without access to downstream browsing tasks, task-specific goals, or evaluator information, thereby *separating page-state collection from tool-using agents and testing tasks* and reducing the risk of tailoring the generated tool set to particular evaluation trajectories.

3.2.2 TOOL GENERATION AND VERIFICATION

After collecting $\mathcal{S}_g$, we develop a tool-generation agent skill that directs a coding agent through three steps (Appendix A.1.2). Given $\mathcal{S}_g$, the agent discovers page-native capabilities, formulates them as candidate tool definitions, and then implements each candidate according to its definition. The candidates are independently verified.

Discover Capabilities. The skill directs the coding agent to inspect the supplied page states, including their DOM structure, JavaScript logic, and visual elements, to identify page-native capabilities and the interaction contexts in which they are applicable. The discovered capabilities and applicability conditions are then used to instantiate the tuple of each candidate tool.

Specify Tools. For each discovered capability, the skill directs the agent to instantiate a candidate tool tuple according to Eq. 1. Each tool $t = (n, d, \Sigma, \mathcal{A}, C)$ is designed as a *single-function primitive*: it implements one bounded application capability rather than combining multiple functions or embedding higher-level agentic reasoning. We further specify three classes of low-level operations that may be composed within a tool:

- **Information extraction:** transforms relevant page information into a structured representation.
- **In-page interaction:** programmatically interacts with elements in the current document, such as filling input fields or activating controls.
- **Navigation:** triggers a transition to a new page state within the same website domain.

Although a tool may compose multiple low-level operations, they must collectively implement a single bounded application capability. For example, a tool for submitting a pull request form may fill several fields and activate the submission control while still representing one semantic function. If tool execution triggers a full-document navigation, execution terminates at that boundary and subsequent interaction with the destination document is returned to the agent. This prevents tool logic from subsuming higher-level reasoning across document contexts and preserves intermediate decision points, allowing the agent to observe the resulting state and determine the next action itself.

Implement Tools. The skill then directs the coding agent to implement the specified tools using JavaScript callbacks for imperative tools and annotated HTML forms for declarative tools. Implementations may compose page-context JavaScript, DOM interaction and extraction, existing application functions, form submission, HTTP requests, and navigation. To preserve application behavior, tools must respect existing permissions, validation, and user-confirmation mechanisms, reflect state-changing outcomes in the shared UI, and return a structured JSON value reporting execution-time evidence, such as success or failure, input validity, extracted information, and observed state changes. For full-document navigation, the return describes the initiated operation; the agent independently observes the destination page.

Verify tools. Tool generation may produce candidates that are redundant, semantically ambiguous, or inconsistent with our design requirements. We therefore develop a separate verification skill that directs an independent agent to validate each generated tool before deployment (Appendix A.1.3). Verification covers four dimensions: semantics, scope, execution, and task-independent selection, using the draft API, developer guidance, observed browser behavior, and our additional design rules.

- **Semantics:** inspect the tool definition $t = (n, d, \Sigma, \mathcal{A}, C)$ to verify that its name, description, input schema, and applicability conditions are internally consistent and follow the design rules.
- **Scope:** enumerate relevant page states and verify that the tool is registered exactly when its applicability condition C_t is satisfied.

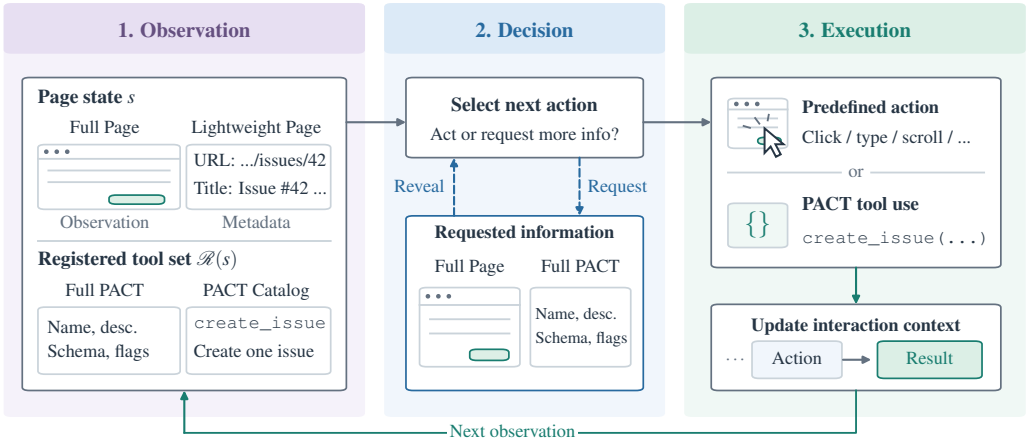

Figure 2: **Agent-side PACT integration.** Two information-exposure choices control how the page information and registered tools are exposed in the agent loop. The tool return value is appended to the context for the next observation.

- **Execution:** exercise both valid and invalid inputs in isolated environments and verify that the observed transition $(s, x) \xrightarrow{t} (s', r)$ conforms to the behavior specified by d, including the resulting page state and tool return value.
- **Selection:** issue task-independent, single-function requests to an LLM and verify that the tool can be correctly selected and invoked from its agent-facing definition.

Any failure triggers revision of the corresponding tool followed by reverification. Only candidates that pass all four checks are included in the verified tool set $\mathcal{T}_v$ and deployed on the website.

3.3 PACT Agent Integration

We integrate PACT into the existing agent–environment interaction loop following the WebMCP API as illustrated in Figure 2. Unlike approaches that largely expose a fixed website-level tool set to the agent up front, PACT couples tool exposure to the evolving page state. At each interaction step, the currently applicable tools are dynamically registered according to the live UI, so the agent only receives tools relevant to the current context. A web agent then proceeds through three stages: *observation*, where it extracts key information from the current page state s and registered tool set $\mathcal{R}(s)$; *decision*, where it can request additional information or select either an exposed tool or a predefined action; and *execution*, where the selected action is carried out and produces a tool return value r or a predefined action's potential output, together with a potentially updated page state s'. The resulting state change may in turn modify the set of tools $\mathcal{R}(s')$ exposed at the next step.

For the *observation* stage, we vary two information-exposure choices: full page vs. lightweight page metadata, and full PACT definitions vs. a compact tool catalog. This yields 4 strategies:

- **Full Page + Full PACT:** the full page observation and definitions of the currently registered tools.
- **Lightweight Page + Full PACT:** lightweight page metadata and full definitions of the currently registered tools, with the standard page observation available on demand.
- **Full Page + PACT Catalog:** the standard page observation and a compact catalog of the currently registered tools, with full tool definitions available on demand.
- **Lightweight Page + PACT Catalog:** lightweight page metadata and a compact tool catalog, with both the standard page observation and full tool definitions available on demand.

For the *decision* stage, the agent may request any information withheld by the selected observation strategy before its next action (e.g., request full PACT when only PACT catalog is provided). For the *execution* stage, the agent can then select either a predefined action or invoke a registered tool t with structured input arguments x conforming to its schema Σ. The resulting tool return value r and page-state changes are returned to the agent, and the set of registered tools is updated accordingly.

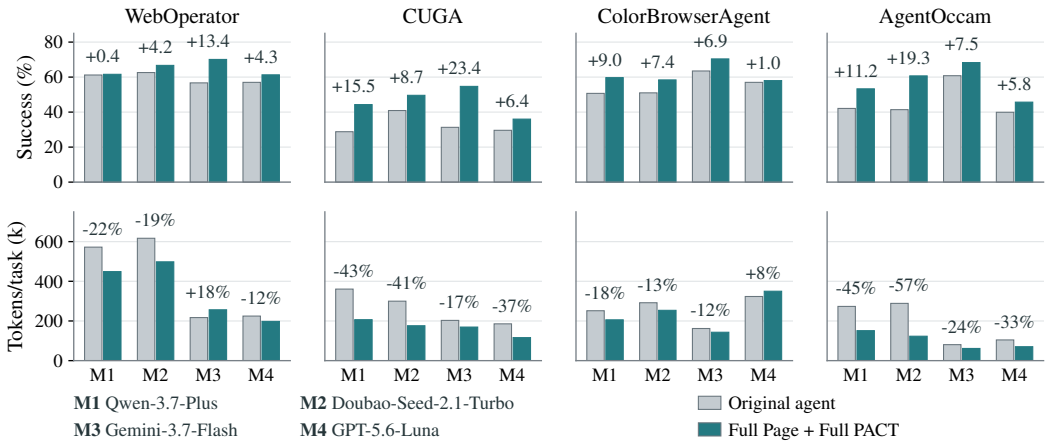

Figure 3: **Integration of PACT into existing agents.** Across four agents × four models (16 configurations) on 812 tasks of WebArena, Full Page + Full PACT raises the success rate (top) for every pair and lowers mean token use (bottom) for 14 of them.

4 Experiments

Dataset and Evaluation. We conduct our main evaluation on WebArena (Zhou et al., 2024), which contains 812 tasks in locally hosted web environments spanning shopping, content management, social forums, collaborative software development, maps, and encyclopedia resources. In Section 4.1.6, we further validate our findings on WebArena Verified (hattami et al., 2025). Appendix A.12 introduces an additional WebArena-style benchmark constructed by us, comprising four mock Chinese websites and 90 tasks across social media, video platforms, and ticket purchasing. All experiments use pass@1 evaluation. We cap single-site tasks at 40 steps and multi-site tasks at 80 steps. For WebArena, we use the original grader, with Qwen3.7-max for LLM-based checks. Automatic reruns are allowed only for failures caused by external errors, such as API timeouts. Appendix A.6 provides the full trajectory admission protocol.

Agents and models. The main experiment includes four agents: WebOperator (Dihan et al., 2025), CUGA (Marreed et al., 2025), ColorBrowserAgent (Wang et al., 2026), and AgentOccam (Yang et al., 2025). Each agent is evaluated with the same four models: Qwen-3.7-Plus, Doubao-Seed-2.1-Turbo, Gemini-3.7-Flash, and GPT-5.6-Luna. We set Full Page + Full PACT as the default tool integration, holding the agent and model fixed within each pair.

Generation and Validity Controls. We place all required materials and generation skills in isolated sandboxes. GPT-5.5 is prompted to generate the PACT tools following Section 3.2 and Appendix A.1, yielding 184 tools in total (Appendix A.3). To guard against test-task leakage and against contamination from any material outside the supplied generation inputs, human reviewers and an AI agent independently audit each verified tool set (Appendix A.2).

4.1 Results and Analysis

We investigate: **(1)** whether PACT improves agents across different backend models; **(2)** how toolset generation and agent-specific optimization affect performance; **(3)** how different tool–agent integration designs affect performance; **(4)** whether the benefits of PACT persist across model sizes; **(5)** whether PACT remains effective for models fine-tuned on web-agent tasks; and **(6)** whether its benefits persist with improved task descriptions.

4.1.1 Effectiveness Across Agents and Models

This experiment tests whether the same PACT set improves task success and resource use across different agents and models. Figure 3 compares the Original agent with Full Page + Full PACT across four agents and four models on WebArena. Success rate increases in all 16 configurations;

Table 1: **Paired-change attribution with PACT integration.** AgentOccam with Qwen-3.7-Plus on 812 WebArena tasks; the tool condition uses Full Page + Full PACT. Counts follow Table 15.

(a) Success changes

Change	Action efficiency	Action selection	Parameter handling	Information access	State handling	Prior-knowledge	Other
Regressions	9	13	5	11	0	0	7
Gains	28	26	28	25	11	0	18

(b) Token changes

Change	Action count	Token / action	Other
Increases	99	215	13
Decreases	399	85	1

(c) Step changes

Change	Tool use	Predefined actions	Longer success	Earlier failure	Other
Increases	118	22	19	0	3
Decreases	0	451	0	25	0

Table 2: **Complete tool-set generation and specialization metrics.** AgentOccam with Qwen-3.7-Plus is evaluated across different tool-set configurations on WebArena. Tool conditions use Full Page + Full PACT. Resource means cover all tasks (all) or each condition's successful tasks (succ.). Throughout the result tables, SR subscripts show percentage-point differences from the matched Original agent, computed from task-level success counts.

Configuration	SR (%)	Tokens		Output tokens		Cache hit (%)		Steps		Tool executions	
		all	succ.	all	succ.	all	succ.	all	succ.	all	succ.
Original agent	42.1	273.4k	105.2k	18.2k	6.7k	12.4	13.9	17.0	8.0	N/A	N/A
Naive tool set	$43.1_{+1.0\%}$	243.0k	121.8k	14.6k	7.4k	12.6	13.3	16.0	8.4	6.4	3.1
PACT	$53.3_{+11.2\%}$	151.2k	57.8k	9.7k	3.7k	16.2	17.9	9.9	4.8	4.2	3.1
PACT (agent-optimized)	$61.6_{+19.5\%}$	70.6k	45.4k	4.4k	2.9k	16.7	17.5	5.1	3.7	3.1	2.6

mean per-task tokens decrease in 14 and steps in 13 out of 16 configurations. Appendix A.7 reports the complete metrics and paired Qwen-3.7-Plus outcomes.

To examine these changes at the task level, we analyze AgentOccam with Qwen-3.7-Plus using the main tool set under Full Page + Full PACT, paired with the same agent–model baseline (Table 1). This condition turns 136 baseline failures into successes, while 45 baseline successes become failures. We group success gains and regressions by *action efficiency* (completing operations within the action budget), *action selection* (choosing the operation or target), *parameter handling* (supplying appropriate values), *information access* (obtaining the required information), *state handling* (establishing or recovering the required page state), and *prior knowledge* (answering from prior knowledge or estimates after retrieval fails); remaining factors enter Other. Token changes are separated into action-count and token-per-action contributions, while step changes distinguish tool use, predefined actions, longer successful runs, and earlier failures. Each changed task receives one category per metric; Appendix A.8.2 gives the definitions and counting procedure.

In this condition, success gains come from improved action efficiency, selection, parameter handling, information access, and state handling. Token reductions mainly result from fewer actions, while increases more often reflect higher token use per action. Most step reductions come from fewer predefined actions, so resource savings should be interpreted together with task success.

4.1.2 Tool-Set Generation and Further Optimization

To examine how tool-set generation and agent-specific optimization affect performance, we construct two additional variants. First, we generate a *Naive tool* set using only the WebMCP API and guidance documents, without our PACT generation skills. Second, we optimize the main PACT tool set for AgentOccam with Qwen-3.7-Plus under the Full Page + Full PACT setting. For each tool and its registered page states, we evaluate the agent–model pair on corresponding single-function tasks and refine the tool name n and description d to improve tool understanding, selection, and invocation. We denote this variant as PACT (*agent-optimized*). Appendix A.8 provides the setup of Naive tool-set generation and task-level attribution analysis.

The two new tool sets are evaluated using AgentOccam with Qwen-3.7-Plus on WebArena as in Table 2. Relative to the Naive tool set, PACT raises success from 43.1% to 53.3% and reduces mean

Table 3: **Complete integration-study metrics.** Different PACT integrations of ColorBrowserAgent are evaluated with Qwen-3.7-Plus and Gemini-3.7-Flash over the 812 WebArena tasks.

Condition	SR (%)	Tokens		Output tokens		Cache hit (%)		Steps		Tool executions	
		all	succ.	all	succ.	all	succ.	all	succ.	all	succ.
Qwen-3.7-Plus											
Original agent	50.7	251.6k	223.3k	13.6k	10.3k	18.4	17.0	9.8	7.8	N/A	N/A
Full Page + Full PACT	59.7 +9.0%	205.8k	171.7k	10.7k	8.5k	19.4	19.2	7.6	6.4	1.8	1.6
Lightweight Page + Full PACT	59.9 +9.1%	156.1k	109.7k	13.0k	9.4k	43.2	44.8	7.9	6.1	3.5	2.9
Full Page + PACT Catalog	57.6 +6.9%	271.3k	206.1k	13.6k	9.6k	22.2	21.9	9.6	7.4	1.3	1.1
Lightweight Page + PACT Catalog	57.6 +6.9%	177.6k	139.8k	14.5k	11.3k	46.1	48.4	8.6	7.1	2.7	2.5
Gemini-3.7-Flash											
Original agent	63.5	162.0k	127.0k	3.0k	2.2k	9.6	8.8	7.6	6.0	N/A	N/A
Full Page + Full PACT	70.4 +6.9%	143.2k	114.6k	2.4k	1.8k	11.8	11.2	5.7	4.6	2.3	2.0
Lightweight Page + Full PACT	65.6 +2.1%	105.5k	73.6k	3.1k	2.2k	6.5	5.5	6.7	5.2	3.3	2.8
Full Page + PACT Catalog	68.5 +4.9%	193.4k	145.1k	3.9k	2.8k	6.6	4.7	7.7	5.8	2.3	1.9
Lightweight Page + PACT Catalog	65.9 +2.3%	132.9k	96.3k	3.7k	2.8k	3.5	3.0	7.5	5.9	3.5	3.1

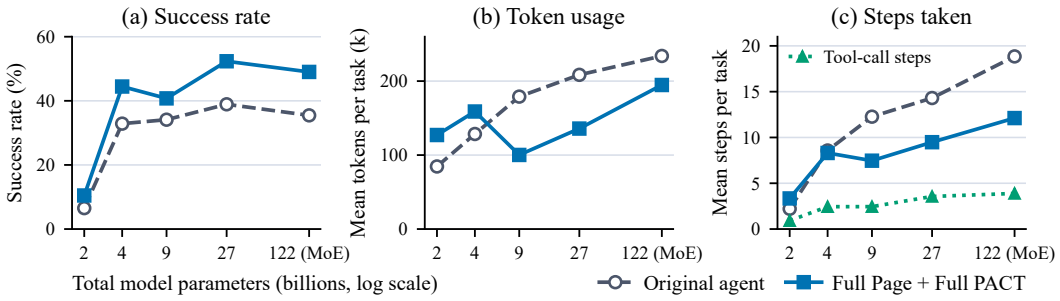

Figure 4: **Model-size trends.** AgentOccam with models in the Qwen-3.5 family on WebArena.

tokens from 243.0k to 151.2k, demonstrating its advantage. The Naive tool set improves success over the original agent baseline by only 1.0 pp despite more tool executions than PACT. The agent-optimized PACT reaches 61.6% and 70.6k tokens, showing pair-specific headroom.

4.1.3 Integration and Screenshot-Ablation Study

We examine how page observations and tool descriptors should enter the agent loop. With ColorBrowserAgent and the main tool set fixed, we cross default versus Lightweight page observations with full descriptors versus a tool catalog for Qwen-3.7-Plus and Gemini-3.7-Flash. Each condition uses the same 812 tasks. Table 3 reports the integration metrics.

Full Page + Full PACT is within 0.2 percentage points of the best design for Qwen-3.7-Plus and achieves the highest success for Gemini-3.7-Flash. Lightweight Page reduces tokens but can sacrifice success, particularly for Gemini-3.7-Flash with Full PACT. PACT Catalog does not guarantee savings: with default page observations, it uses more tokens than full descriptors for both models.

Because PACT tools are implemented entirely through code, we further test whether visual input remains necessary by removing screenshots while retaining textual page observations and PACT tools. Table 17 in Appendix A.9 reports these results. Removing screenshots improves Qwen-3.7-Plus but degrades Gemini-3.7-Flash under both matched integration settings: with Full Page + Full PACT, success changes from 59.7% to 65.6% for Qwen-3.7-Plus, but from 70.4% to 57.4% for Gemini-3.7-Flash. This indicates that the effect of visual input is model-dependent. Together, these results show that page observations and tool descriptions affect both task success and token usage.

4.1.4 Model-Size Study

Figure 4 compares the Original agent and Full Page + Full PACT across Qwen-3.5 model sizes using a fixed AgentOccam implementation on WebArena. PACT improves success across all evaluated

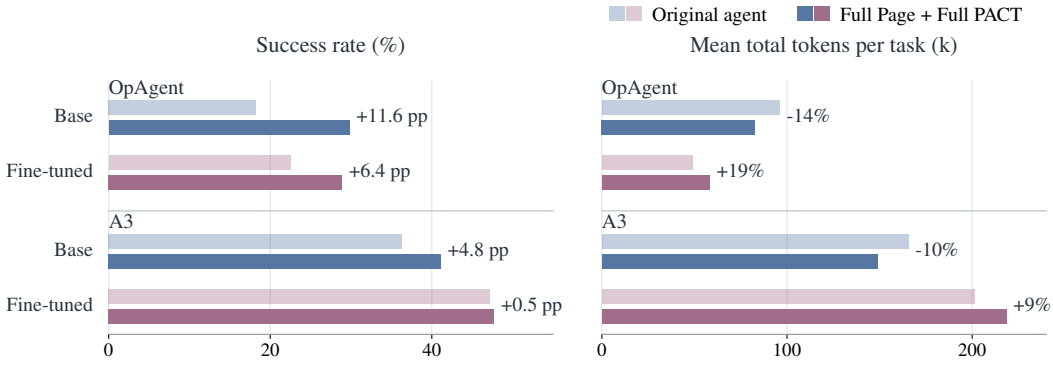

Figure 5: **Tool-use benefits before and after fine-tuning.** Paired horizontal bars compare the Original agent with Full Page + Full PACT on WebArena. Labels show success-rate changes in percentage points and relative changes in all-task mean tokens.

Table 4: **Results of AgentOccam with Doubao-Seed-2.1-Turbo on WebArena Verified.** Successful-task means (succ.) use each condition's own strict-success subset. Five tasks in the Original agent condition and six in Full Page + Full PACT have evaluation error failures.

Condition	SR (%)	Tokens		Steps		Tool executions	
		all	succ.	all	succ.	all	succ.
Original agent	33.5	241.0k	86.9k	12.2	7.3	N/A	N/A
Full Page + Full PACT	41.7$_{+8.3\%}$	143.2k	69.5k	7.8	5.5	3.6	2.4

sizes, but increases token usage for the 2B and 4B models, suggesting that smaller models may incur additional overhead when leveraging the tools. Detailed results are reported in Appendix A.10.

4.1.5 Fine-Tuning and Tool-Use Benefits

We compare OpAgent-32B and A3-Qwen3.5-9B with their respective base models on the original OpAgent and A3 frameworks to examine whether web-agent fine-tuning strengthens the benefit of PACT. As shown in Figure 5, PACT's success gain decreases after fine-tuning, from 11.6 to 6.4 points for OpAgent and from 4.8 to 0.5 for A3. Token usage decreases for both base models but increases for both fine-tuned models, suggesting that fine-tuning and PACT do not provide simply additive gains. The small A3 improvement remains descriptive without paired uncertainty estimates. Complete metrics are reported in Appendix A.11.

4.1.6 Supplementary Validity Evaluation

As a supplementary validity check, we evaluate AgentOccam with Doubao-Seed-2.1-Turbo on 812 WebArena Verified tasks (hattami et al., 2025). As shown in Table 4, PACT improves success by 8.3 percentage points while reducing all-task token usage by 40.6% and steps by 36.3%. We use the official WebArena Verified v1.2.3 automatic evaluator and count evaluation errors as failures without manual overrides. These results extend the observed benefits of PACT to WebArena Verified.

5 Conclusion

We introduce **PACT**, an application-grounded framework for generating and integrating independently verified, page-coupled WebMCP tools. PACT derives bounded single-function capabilities from page evidence, dynamically registers them according to the live page state, and leaves higher-level planning and subsequent decisions to the calling agent. Across 16 WebArena agent–model configurations, PACT improves success in all cases and reduces mean token usage in 14. Further studies show that these gains depend on tool quality, information exposure, and model choice, motivating joint design of tool generation and integration.

AI USE STATEMENT

We, human researchers, developed the procedural AI agent skills used in this paper and refined the wording in these skills with AI assistance.

AI coding agents, LLM responses, and web agents were used to:

- Refine the writing and figures.
- Refine the wording in our agent skills.
- Assist page state selection and task-independence audit based on our AI agent skills.
- Generate and verify PACT tools based on our AI agent skills.
- Integrate PACT tools into web agents based on our AI agent skills.
- Monitor experiments for disconnections and external errors.
- Generate websites for an additional experiment included only in the Appendix.

We manually reviewed manuscript changes, selected page states, audit results, tool definitions, tool implementations, tool verification results, PACT integrations on the agent-side, experiment logs, result data transformations, and generated websites.

We checked cited sources on Google Scholar, supervised all experiments, monitored all AI generations, verified experiment results against the recorded trajectories and evaluator outputs, and take responsibility for the final content of this work.

ETHICS STATEMENT

The experiments use locally-hosted WebArena instances and generated mock websites; they do not involve human subjects, private user data, live purchases, or any other information that should not be published.

REPRODUCIBILITY STATEMENT

The main text describes the tool-generation and agent-integration methods, experimental setups, evaluators, and reporting metrics. The appendix provides agent skills used, verification procedures, tool statistics, and execution details.

We plan to release the experimental setup manifests, PACT tool sets, agent-side PACT integrations, and processed experiment results publicly after publication.

REFERENCES

Anthropic. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol, 2024.

André Cipriani Bandarra and Alexandra Klepper. When to use WebMCP and MCP. https://developer.chrome.com/docs/ai/webmcp/compare-mcp, 2026. Published March 11, 2026; updated May 19, 2026; snapshot accessed August 18, 2026.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Mahir Labib Dihan, Tanzima Hashem, Mohammed Eunus Ali, and Md Rizwan Parvez. WebOperator: Action-aware tree search for autonomous agents in web environment. *arXiv preprint arXiv:2512.12692*, 2025.

Yuyu Guo, Wenjie Yang, Siyuan Yang, Ziyang Liu, Cheng Chen, Yuan Wei, Yun Hu, Yang Huang, Guoliang Hao, Dongsheng Yuan, et al. OpAgent: Operator agent for web navigation. *arXiv preprint arXiv:2602.13559*, 2026.

Amine El hattami, Megh Thakkar, Nicolas Chapados, and Christopher Pal. WebArena Verified: Reliable evaluation for web agents. In *Workshop on Scaling Environments for Agents*, 2025. URL https://openreview.net/forum?id=94tlGxmqkN.

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

Lawrence Keunho Jang, Jing Yu Koh, Daniel Fried, and Ruslan Salakhutdinov. Odysseys: Benchmarking web agents on realistic long horizon tasks. *arXiv preprint arXiv:2604.24964*, 2026.

Hongrui Jia, Jitong Liao, Xi Zhang, Haiyang Xu, Tianbao Xie, Chaoya Jiang, Ming Yan, Si Liu, Wei Ye, and Fei Huang. OSWorld-MCP: Benchmarking MCP tool invocation in computer-use agents. In *International Conference on Learning Representations*, volume 2026, pp. 63734–63768, 2026.

Alexandra Klepper. WebMCP best practices. https://developer.chrome.com/docs/ai/webmcp/best-practices, 2026a. Published May 18, 2026; snapshot accessed August 18, 2026.

Alexandra Klepper. WebMCP. https://developer.chrome.com/docs/ai/webmcp, 2026b. Published May 18, 2026; updated August 7, 2026; snapshot accessed August 18, 2026.

Alexandra Klepper. How WebMCP fits in user journeys. https://developer.chrome.com/docs/ai/webmcp/use-cases, 2026c. Published May 18, 2026; snapshot accessed August 18, 2026.

Alexandra Klepper and François Beaufort. Declarative API. https://developer.chrome.com/docs/ai/webmcp/declarative-api, 2026a. Published May 18, 2026; snapshot accessed August 18, 2026.

Alexandra Klepper and François Beaufort. Imperative API. https://developer.chrome.com/docs/ai/webmcp/imperative-api, 2026b. Published May 18, 2026; updated July 1, 2026; snapshot accessed August 18, 2026.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. 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)*, pp. 881–905, 2024.

Kasper Kulikowski. Test WebMCP with Evals. https://developer.chrome.com/docs/ai/webmcp/evals, 2026. Published May 19, 2026; updated May 28, 2026; snapshot accessed August 18, 2026.

Mugeng Liu, Shuoqi Li, Yixuan Zhang, and Yun Ma. Beyond static endpoints: Tool programs as an interface for flexible agentic web services. *arXiv preprint arXiv:2606.19992*, 2026.

Xiao Liu, Tianjie Zhang, Yu Gu, Iat Long Iong, XiXuan Song, Yifan Xu, Shudan Zhang, Hanyu Lai, Jiadai Sun, Xinyue Yang, et al. VisualAgentBench: Towards large multimodal models as visual foundation agents. In *International Conference on Learning Representations*, volume 2025, pp. 95650–95707, 2025.

Renze Lou, Baolin Peng, Wenlin Yao, Qianhui Wu, Hao Cheng, Suman Nath, Wenpeng Yin, and Jianfeng Gao. The tool illusion: Rethinking tool use in web agents. *arXiv preprint arXiv:2604.03465*, 2026.

Xing Han Lù and Siva Reddy. Structured distillation of web agent capabilities enables generalization. *arXiv preprint arXiv:2604.07776*, 2026.

Yadong Lu, Lingrui Xu, Chao Huang, and Ahmed Awadallah. Webwright: A terminal is all you need for web agents. https://github.com/microsoft/Webwright, 2026. GitHub repository.

Sami Marreed, Alon Oved, Avi Yaeli, Segev Shlomov, Ido Levy, Offer Akrabi, Aviad Sela, Asaf Adi, and Nir Mashkif. Towards enterprise-ready computer using generalist agent. *arXiv preprint arXiv:2503.01861*, 2025.

Julia Pagnucco and Alexandra Klepper. WebMCP tool security. https://developer.chrome.com/docs/ai/webmcp/secure-tools, 2026. Published June 9, 2026; updated July 1, 2026; snapshot accessed August 18, 2026.

Viraj Prabhu, Yutong Dai, Matthew Fernandez, Krithika Ramakrishnan, Jing Gu, Yanqi Luo, Silvio Savarese, Caiming Xiong, Junnan Li, Zeyuan Chen, and Ran Xu. WALT: Web agents that learn tools. In *International Conference on Learning Representations*, volume 2026, pp. 55589–55607, 2026.

Jihong Wang, Jiamu Zhou, Weiming Zhang, Teng Wang, Weiwen Liu, Zhuosheng Zhang, Xingyu Lou, Weinan Zhang, Huarong Deng, and Jun Wang. ColorBrowserAgent: Complex long-horizon browser agent with adaptive knowledge evolution. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pp. 665–680, 2026.

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

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

Web Machine Learning Community Group. WebMCP. https://webmachinelearning.github.io/webmcp/, 2026. Draft Community Group Report, August 17, 2026; snapshot accessed August 18, 2026.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. BrowseComp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025.

Zikai Xiao, Jianhong Tu, Chuhang Zou, Yuxin Zuo, Zhi Li, Peng Wang, Bowen Yu, Fei Huang, Junyang Lin, and Zuozhu Liu. WebWorld: A large-scale world model for web agent training. *arXiv preprint arXiv:2602.14721*, 2026.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? assessing the current state of web agents. *arXiv preprint arXiv:2504.01382*, 2025.

Ke Yang, Yao Liu, Sapana Chaudhary, Rasool Fakoor, Pratik A Chaudhari, George Karypis, and Huzefa Rangwala. AgentOccam: A simple yet strong baseline for LLM-based web agents. In *International Conference on Learning Representations*, volume 2025, pp. 97533–97565, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

A APPENDIX

The appendix first supplies generation and evaluation details (Sections A.1–A.6), then complete results in the order of the main studies (Sections A.7–A.11). The separate Chinese-site extension follows in Section A.12.

A.1 PACT TOOL GENERATION: SKILLS AND PROMPTS

The prompts and skills we developed for each tool generation stage are detailed below. Bracketed fields identify the local file or folder paths. The local copy of WebMCP API draft comes from the WebMCP Draft Community Group Report (Web Machine Learning Community Group, 2026). The local copy of WebMCP Best Practices snapshot comes from (Klepper, 2026a).

Before each stage, we create a separate sandbox where only the skills, required reference guides, permitted documents and code, and necessary environment access are provided. We transfer only the specified outputs between stages and keep benchmark-related artifacts outside generation sandboxes.

A.1.1 STAGE 1: PAGE-STATE COLLECTION

The page state collection skill is designed to perform BFS on a given website to collect targeted page states for generation. No browsing tasks are provided at this stage.

Prompt 1

`/collect-page-states` [base URL of the website hosted locally]

Skill 1: Independent page state collection
`/collect-page-states`

Role and inputs.
Starting from this base URL, collect non-repetitive, interaction-reachable page states of this website. The user should log in to the website before invoking this skill if possible.
Procedure.
Explore from this base URL using Breadth-First Search to construct a navigation tree of this website. Start with the initial page state of base URL in the BFS queue.
For each page state in queue,

1. Label the state with its metadata and take a screenshot for reference.
2. Save a snapshot of the page state that fixes the authentication, fixture, viewport, locale, and foreground conditions needed to reproduce the page state.
3. Normalize the state by site route pattern, authentication role, mounted tab, modal or panel, viewport class, neutral fixture class, and foreground assumption.
4. Add this state and the transition to this state into the output collection.
5. Inspect the following interaction on current page state that may lead to a new page state within the current website's domain:
   (a) Hyperlinks that may load an entirely new URL and document;
   (b) Form submissions via a button;
   (c) JavaScript that update the URL and view dynamically;
   (d) Tabs, accordions, or filter buttons changes the local component or URL query parameters to show different content.

   Add a state to the BFS queue when its DOM structure, UI conditions, or application state differs from the collected states, even if the URL is unchanged. Mark states with the same DOM structure but different URLs as similar states for further inspection during generation.

Stop when no new state can be reached. Avoid looping in exploration.
Outputs.
A collection of page states reachable from the supplied homepage within the website's domain.

A.1.2 STAGE 2: TOOL GENERATION

Prompt 2

`/tool-authoring`
Generate WebMCP tools for the selected page states of the website in [sanitized page state collection]. You can access the locally hosted original website at [site local URL] with the browser configuration specified in [browser configuration file]. For the WebMCP API, use [the local copy of WebMCP API draft]. For WebMCP best practices, refer to [the local copy of WebMCP Best Practices snapshot].

Skill 2: Page-coupled tool generation
`/tool-authoring`

Role and inputs.
Generate tools on the provided page-state collection based on live page evidence, page source code, and WebMCP guidance. Consult the WebMCP API draft provided for interface semantics and Google's developer guidance provided for tool design. Do not use any benchmark tasks, expected actions, answers, or evaluators to determine tool behavior.
Discover capabilities.
Inspect each supplied state, including its DOM, application logic, visible controls, authentication assumptions, and before/after state signals. Record existing tools and uncertain observations. Consider structured reads, stable list/action pairs, forms, stateful widgets, and fragile repeated interactions. Reject duplicate wrappers, pure reasoning or summarization, login/setup helpers, task-specific shortcuts, and operations that bypass the page's validation or leave its UI stale. Record useful operations that remain predefined actions.
Specify tools.
Specify the name, description, input schema, annotations, registration conditions, execution behavior, and expected returns before implementation. Keep each tool a single page-native function. Separate listing from acting on a selected item, and use stable identifiers rather than row positions. Obtain identifiers already implied by the current page from live page context. Derive dynamic choices from current application state rather than task-specific constants. For a wizard, preserve stable user-confirmable checkpoints as separate operations.
Implement tools.
Each tool can execute one or more following operations:

- **Information extraction**: transforms relevant information into a structured format.
- **In-page interaction**: programmatically interacts with elements in the current document.
- **Navigation**: navigates to a new page state within the website domain.

The implementation should follow these rules:

1. Use a native declarative form when it represents the operation. If a real form-like workflow lacks a usable HTML form, a synthetic form may declare parameters and receive invocation, but must drive the existing controls, validation, and review path. Use imperative callbacks for other capabilities.
2. Register only when the required controls and interaction context are present. Remove registrations when the relevant route, component, modal, or tab scope ends. Resolve live state at invocation time; avoid stale DOM captures and hidden duplicate controls.
3. Preserve permissions, confirmation, validation, and durable-state checks. Keep the UI consistent with the observed outcome. Return one structured value; expected failures return an error with recovery information. Separate trusted status fields from untrusted page content and set annotations to reflect actual behavior.
4. When a document-changing navigation destroys the execution context, validate inputs and return before deferred navigation. Do not continue the operation in the destination document or claim its final state was verified. When context survives, wait for observable state and check that optimistic updates were not rejected or reverted.

Output and handoff. Provide definitions and implementations, page evidence, rejected candidates, registration/lifecycle rules, expected returns, known exceptions, and verification targets. Mark static evidence that still requires live verification.

A.1.3 STAGE 3: TOOL VERIFICATION

Prompt 3

`/tool-verification`
Verify the generated WebMCP tools in [candidate definitions and implementations] with evidence in [task-neutral page evidence and expected returns]. You can access the locally hosted website at [local URL] with the browser configuration specified in [browser configuration file]. For the WebMCP API, use [the local copy of WebMCP API draft]. For WebMCP best practices, refer to [the local copy of WebMCP Best Practices snapshot].

Skill 3: Independent tool verification
`/tool-verification`

Role and inputs.
In a context separate from tool generation, inspect the tool definitions and implementations, page evidence, expected returns, and target runtime. Use task-independent requests for selection checks; do not use benchmark tasks, reference answers, scoring rules, or task-level results.
Checks for each candidate.

1. **Semantics.** Check agreement among the description, schema, implementation, structured return, error behavior, and annotations. Check stable identifiers, parameter necessity, page-owned affordances, and preserved validation/review paths. Do not assume that a structured return implies a registered output schema.
2. **Scope.** Enumerate tools in the target browser. Test states where the tool should be present and absent, including relevant route, modal, tab, authentication, and control changes. Check cleanup, re-registration, duplicate registration, and selectors against active rather than hidden UI.
3. **Execution.** Invoke the tool with valid and invalid inputs. Compare reads with rendered fields. For list/action pairs, select an identifier returned by the list and inspect the subsequent action's effect. For mutations and navigation, compare the explicit return with independently observed DOM, route, and durable application state. Test missing controls and invalid handles for recoverable errors. Do not accept an optimistic success message as proof.
4. **Selection.** Construct task-independent natural-language requests grounded in the page capability. Record the starting state, expected tool, and argument constraints, including order when multiple calls matter. Check whether an LLM selects and invokes the intended tool from its definition among relevant alternatives.

Diagnosis and revision.
Distinguish implementation or selection failures from stale runtime, fixture drift, missing browser support, and verification-harness errors. Report the violated requirement, observation, cause, and smallest justified revision. Repeat affected checks after revision. Do not infer live correctness from static inspection alone.
Outputs.
Per-tool verdicts: `PASS`, `FAIL`, `VERIFY LIVE`, or `ENV BLOCKED`. Preserve calls, sample success and error returns, tool lists before/after execution, observed page state, selection results, and evidence gaps. Only candidates passing every required check enter the verified set; unresolved checks remain non-passes.

A.2 TASK-INDEPENDENCE AUDIT

Human reviewers and an AI agent independently audit the verified tool sets following the protocol detailed below to prevent the leakage of benchmark tasks.

Prompt 4

`/task-independence-audit`
Audit the fairness of generated WebMCP tools in [verified tool source/catalog] using [original task list], [sanitized page-state collection], and [generation/verification provenance].

Skill 4: Independent task-independence audit
`/task-independence-audit`

Role and inputs.
Perform a read-only audit of generated WebMCP tools in a separate context. Inspect the provided benchmark task list, page state collection, final tool source and catalog, page evidence, and generation/verification access records.
Procedure.

1. Check access provenance first. Review available context manifests, file permissions, file reads, browser history, tool calls, and inherited conversation context. Determine whether generation or verification accessed task corpora, task analyses, answers, evaluator rules, task-derived prompts, or task-level results.
2. Compare the final tools with the tasks. Inspect names, descriptions, schemas, enums, defaults, examples, hardcoded values, returned fields, registration conditions, and execution boundaries for task-specific design.
3. Require each tool to be independently justified by page affordances, operate on live data, and remain useful across neutral fixtures. Check that scope stays within the supplied page-state collection. Usefulness for a benchmark task alone is not evidence of customization.
4. Return `FAIL` for confirmed forbidden access, task customization, or page-scope escape. Otherwise return `INCONCLUSIVE` if access evidence cannot establish isolation. Return `PASS` only when both provenance and surface checks pass. Absence of task strings is not proof of isolation.
5. Do not edit tools or send task-specific patch instructions back to generation. Preserve a failed surface as evidence and regenerate from the sanitized collection in a clean context, followed by verification and a new audit. For inconclusive cases, ask the researchers to double-check the unresolved evidence and assist in determining the final verdict.

Output.
Record input identities, provenance coverage, verdict, affected tools, supporting evidence, and the required recovery boundary. Keep findings separate from future generation contexts.

For `INCONCLUSIVE` cases, we double-check the unresolved evidence to help the agent reach a final verdict. When a tool set receives a final `FAIL` verdict, the entire tool set is discarded. We then restart tool generation from the generation stage, without transferring findings into the new generation sandbox.

A.3 Main PACT Tool Set Statistics

Table 5 summarizes the main PACT tool set used in our experiments. Tools are categorized along two dimensions: API type (declarative or imperative) and functionality (read-only, interaction, or navigation).

Table 5: **Main PACT tool set statistics.** A tool may use both declarative and imperative mechanisms, but is assigned a single primary functional role.

Application	Total	Declarative	Imperative	Read-only	Interaction	Navigation
Reddit	19	2	17	3	9	7
GitLab	69	7	63	19	25	25
Shopping	23	3	21	7	8	8
Shopping Admin	58	7	51	9	34	15
Map	6	1	5	1	0	5
Wikipedia	9	1	8	5	0	4
Overall	184	21	165	44	76	64

A.4 Page-Coupled Tool Examples

Table 6 instantiates the tool tuple from Section 3.1 for three composable Shopping tools covering information extraction, in-page interaction, and navigation.

Table 6: **Three PACT tool specifications.** Examples of information extraction, in-page interaction, and navigation tools from the main PACT tool set.

Name n	`list_products`	`add_to_cart`	`sort_products`
Description d	Reads products from the current category, search, home, or wish-list page.	Adds this product to the cart in the given quantity.	Sorts the current catalog or search grid by a visible sorter field.
Input schema Σ	Empty object; no arguments.	`qty`: quantity field of the current product form; described as a whole number of at least one.	Required `field`: enum of current sorter values; optional `direction`: `asc` or `desc`, default `asc`.
Annotations $\mathcal{A}$	`readOnlyHint: true`	No explicit annotation flags.	`readOnlyHint: false`
Condition C	A supported product-listing page; an empty wish list is excluded.	The current product's add-to-cart form and quantity input are present.	The current listing exposes a sorter with available field options.
Type	Information extraction	In-page interaction	Navigation
API	Imperative JavaScript callback.	Declarative native-form annotations.	Imperative JavaScript callback.
Operations	Read product cards; extract identifiers, names, prices, and available ratings; assemble a structured list.	Fill quantity; submit the form; read confirmation or error messages and cart-counter changes.	Read sorter options; validate the field and direction; construct the listing URL; initiate navigation.
Return value r	Product records, or an empty list with a note.	Page confirmation or cart-count message, or an error if the operation fails or confirmation is unavailable.	Selected field, direction, and navigation status, or an input-validation error.

Table 7 illustrates an action sequence for adding one unit of the lowest-priced suitable product from a listing that exposes price sorting. The product-link target is selected from the observed page after inspecting the returned products. The three tool calls are separate decisions, with a predefined click action connecting the listing to the product-detail form.

Table 7: **Illustrative action sequence.** Selected actions, tool inputs, and return-value contents are shown; the product-link target denotes the link selected by the agent.

Step	Selected action	Tool name n	Tool input x	Return value r
k	PACT tool use	`sort_products`	`{"field":"price", "direction":"asc"}`	Selected sorting options and navigation status.
$k+1$	PACT tool use	`list_products`	`{}`	Structured product attributes for comparison.
$k+2$	Click product link	—	—	—
$k+3$	PACT tool use	`add_to_cart`	`{"qty":1}`	Cart-update confirmation or an error message.

This sequence illustrates several mechanisms underlying the categories in Table 15. Combining related operations within a single invocation can improve *Action efficiency*, while restricting sorting arguments to available UI options and passing quantity through the native form supports *Parameter handling*. Structured product records facilitate comparison (*Information access*), but the agent still determines product suitability and selects the corresponding link (*Action selection*). The resulting page state and tool feedback help the agent assess whether navigation and cart updates succeeded (*State handling*). Together, these mechanisms can reduce low-level action decisions and repeated interpretation of page content, potentially lowering reasoning demands and token use.

A.5 Tool Execution Boundaries Compared with WALT

Table 8 compares implementations of four matched website functions in WALT (Prabhu et al., 2026) and our main PACT tool set.

Search and price editing tools in PACT illustrate separate caller decisions between operations that WALT bundles. Submission and issue creation retain a similar semantic scope: the distinction is their page-dependent availability and execution boundary.

Table 8: Matched tool functions and execution boundaries. WALT can bundle navigation and subsequent interactions within one invocation. Our tools expose functions in the current page context and leave tool composition and subsequent decisions to the calling agent.

Function	WALT	PACT tools
Product search (Shopping)	`search_products` opens the homepage, submits a query, waits for results, and sets sorting and the page limit.	`open_product_search` initiates navigation to search results. `sort_products` is a separate operation on the current product grid.
Price update (Shopping Admin)	`update_product_price` opens a product's edit page, changes its price and optional attributes, then saves.	`set_price` stages a price change on the current edit page; `save_product` commits it in a separate invocation.
Submission creation (Reddit)	`create_submission` navigates to the submission page, fills the form, selects a forum, and submits.	`create_submission` fills and submits the current submission form. The semantic function is similar; the required starting context differs.
Issue creation (GitLab)	`create_issue` opens a configured project's new-issue page, fills the title and description, and submits.	`create_issue` is available with the current project's visible creation control or form. It creates the issue through the application API, returns issue fields, and initiates navigation to the issue.

A.6 Trajectory Admission

All experiments use pass@1. All single-site tasks are capped at 40 steps and all multi-site tasks are capped at 80 steps. Every task runs in an isolated, standalone environment to avoid any possible interference from other tasks or agents' external memory. External-error failures are retried under the same condition until the first valid completion, which is retained regardless of success or failure. Each retry resets the agent and website state and may run in a different batch or on another machine. A task is considered successful if and only if its score is equal to 1. Termination caused by step-limit reached with a score less than 1, internal errors in agents' code, and API requests rejected by the API because of security reasons are counted as failure.

A.7 Complete Main Study Results

Tables 9–12 provide the full metrics for Figure 3, with means over all 812 tasks or each condition's successful tasks. Cache hit is the mean of each task's cached input tokens divided by its input tokens.

Table 9: **Main results for WebOperator.** Pass@1 results on WebArena comparing the Original agent baseline with Full Page + Full PACT. Success rate (SR) and per-task averages of total tokens, output tokens, input cache hit rate, steps, and tool executions are reported over all tasks and each condition's successful-task subset (succ.).

Model	Condition	SR (%)	Tokens		Output tokens		Cache hit (%)		Steps		Tool executions	
			all	succ.	all	succ.	all	succ.	all	succ.	all	succ.
Qwen-3.7-Plus	Original agent	61.2	572.2k	394.5k	22.3k	11.6k	16.1	16.5	13.3	9.2	N/A	N/A
	Full Page + Full PACT	61.6$_{+0.4\%}$	448.9k	328.1k	17.4k	10.7k	16.5	16.6	12.5	8.5	4.9	2.6
Doubao-Seed-2.1-Turbo	Original agent	62.6	616.9k	444.2k	19.5k	12.3k	12.0	11.7	14.8	10.1	N/A	N/A
	Full Page + Full PACT	66.7$_{+4.2\%}$	498.2k	385.7k	16.1k	11.9k	14.7	14.6	11.4	9.0	4.1	3.0
Gemini-3.7-Flash	Original agent	56.7	217.1k	208.6k	5.4k	5.4k	2.6	3.0	6.9	6.8	N/A	N/A
	Full Page + Full PACT	70.1$_{+13.4\%}$	256.4k	208.3k	5.9k	4.8k	4.3	4.5	7.6	6.3	2.6	2.1
GPT-5.6-Luna	Original agent	57.0	224.7k	176.5k	7.1k	4.4k	0.8	0.4	12.2	9.5	N/A	N/A
	Full Page + Full PACT	61.3$_{+4.3\%}$	197.8k	146.3k	5.3k	2.7k	0.7	0.0	12.3	8.3	4.6	2.2

Table 10: **Main results for CUGA.** Pass@1 on WebArena. Conditions and columns follow Table 9.

Model	Condition	SR (%)	Tokens		Output tokens		Cache hit (%)		Steps		Tool executions	
			all	succ.	all	succ.	all	succ.	all	succ.	all	succ.
Qwen-3.7-Plus	Original agent	28.8	360.7k	179.9k	5.0k	3.1k	23.0	18.2	10.5	6.2	N/A	N/A
	Full Page + Full PACT	44.3 +15.5%	206.5k	199.1k	3.8k	2.8k	20.0	18.0	4.9	5.6	0.7	0.6
Doubao-Seed-2.1-Turbo	Original agent	40.9	299.8k	348.3k	3.6k	3.4k	11.7	11.4	8.8	9.8	N/A	N/A
	Full Page + Full PACT	49.6 +8.7%	176.0k	204.5k	2.5k	2.7k	14.4	16.5	5.7	6.7	2.0	2.3
Gemini-3.7-Flash	Original agent	31.3	202.9k	211.0k	1.6k	1.5k	5.2	6.8	5.8	5.9	N/A	N/A
	Full Page + Full PACT	54.7 +23.4%	169.3k	150.6k	1.6k	1.4k	2.4	4.3	4.5	4.2	1.3	1.8
GPT-5.6-Luna	Original agent	29.6	185.5k	150.6k	2.4k	1.7k	5.3	2.2	5.4	6.0	N/A	N/A
	Full Page + Full PACT	36.0 +6.4%	116.1k	107.4k	1.9k	1.3k	7.3	4.9	3.0	3.9	1.6	1.7

Table 11: **Main results for ColorBrowserAgent.** Pass@1 on WebArena. Conditions and columns follow Table 9.

Model	Condition	SR (%)	Tokens		Output tokens		Cache hit (%)		Steps		Tool executions	
			all	succ.	all	succ.	all	succ.	all	succ.	all	succ.
Qwen-3.7-Plus	Original agent	50.7	251.6k	223.3k	13.6k	10.3k	18.4	17.0	9.8	7.8	N/A	N/A
	Full Page + Full PACT	59.7 +9.0%	205.8k	171.7k	10.7k	8.5k	19.4	19.2	7.6	6.4	1.8	1.6
Doubao-Seed-2.1-Turbo	Original agent	51.0	291.9k	270.8k	41.6k	39.6k	19.7	21.0	10.2	8.9	N/A	N/A
	Full Page + Full PACT	58.4 +7.4%	253.7k	190.2k	40.4k	33.3k	23.1	21.7	8.8	7.0	2.4	1.9
Gemini-3.7-Flash	Original agent	63.5	162.0k	127.0k	3.0k	2.2k	9.6	8.8	7.6	6.0	N/A	N/A
	Full Page + Full PACT	70.4 +6.9%	143.2k	114.6k	2.4k	1.8k	11.8	11.2	5.7	4.6	2.3	2.0
GPT-5.6-Luna	Original agent	57.0	323.4k	262.5k	3.8k	2.8k	15.1	15.4	12.4	9.6	N/A	N/A
	Full Page + Full PACT	58.0 +1.0%	350.0k	256.6k	4.0k	2.8k	18.2	17.2	15.5	10.6	2.3	2.1

Table 12: **Main results for AgentOccam.** Pass@1 on WebArena. Conditions and columns follow Table 9.

Model	Condition	SR (%)	Tokens		Output tokens		Cache hit (%)		Steps		Tool executions	
			all	succ.	all	succ.	all	succ.	all	succ.	all	succ.
Qwen-3.7-Plus	Original agent	42.1	273.4k	105.2k	18.2k	6.7k	12.4	13.9	17.0	8.0	N/A	N/A
	Full Page + Full PACT	53.3 +11.2%	151.2k	57.8k	9.7k	3.7k	16.2	17.9	9.9	4.8	4.2	3.1
Doubao-Seed-2.1-Turbo	Original agent	41.4	288.7k	146.5k	17.5k	9.0k	17.9	11.9	17.4	10.7	N/A	N/A
	Full Page + Full PACT	60.7 +19.3%	123.2k	90.4k	7.0k	5.1k	9.1	6.5	8.9	7.3	4.2	3.7
Gemini-3.7-Flash	Original agent	60.8	80.6k	64.0k	3.1k	2.4k	4.6	4.0	8.8	7.6	N/A	N/A
	Full Page + Full PACT	68.3 +7.5%	61.5k	51.7k	1.9k	1.6k	4.4	4.2	6.6	5.8	3.1	2.9
GPT-5.6-Luna	Original agent	39.9	104.4k	66.3k	3.2k	2.2k	1.2	0.6	9.7	7.7	N/A	N/A
	Full Page + Full PACT	45.7 +5.8%	70.3k	51.2k	2.8k	2.4k	39.3	30.1	6.0	5.6	2.5	2.4

Table 13: **Strict outcome transitions for Qwen-3.7-Plus on WebArena.** Gain is Original-agent failure and Full Page + Full PACT success; regression is the reverse. ΔSR is the paired success-rate difference with a 95% normal confidence interval. p is the two-sided exact McNemar test without multiplicity adjustment.

Agent	N	Gain	Regression	Shared success	Shared failure	ΔSR (pp; 95% CI)	p
WebOperator	812	69	66	431	246	$+0.4$ $[-2.4, 3.2]$	.863
CUGA	812	168	42	192	410	$+15.5$ $[12.2, 18.9]$	$< .001$
ColorBrowserAgent	812	115	42	370	285	$+9.0$ $[6.0, 12.0]$	$< .001$
AgentOccam	812	136	45	297	334	$+11.2$ $[8.1, 14.4]$	$< .001$

A.8 Tool-Set Generation Details and Task Examples

Table 2 reports the tool-set comparison; the generation setup and paired-task analysis are detailed below.

A.8.1 Naive Tool Set: Prompt, Setup, and Tool Statistics

The Naive tool set uses a separate prompt, environment, and set of input materials.

Prompt 5

Generate WebMCP tools for the provided locally-hosted websites based on these documents. Reuse the current WebMCP tool injection method.

Independent environment. In a new sandbox, we initialize a fresh Git repository which contains only the supplied documentation archive, original WebArena website container images (not including the WebArena tasks), the existing WebMCP tool-injection implementation, and other necessary runtime environment setups. The generation agent could not access the main tool set or any related documentation or materials, procedural skills, prior generation records, benchmark tasks, or evaluator information.

Supplied materials. The supplied documentation collection, dated August 18, included the August 17 WebMCP API draft (Web Machine Learning Community Group, 2026) used throughout this paper. Chrome for Developers materials covered getting started (Klepper, 2026b), declarative and imperative APIs (Klepper & Beaufort, 2026a;b), and best practices (Klepper, 2026a). Additional documents covered tool security (Pagnucco & Klepper, 2026), evaluation (Kulikowski, 2026), use cases (Klepper, 2026c), and comparison with MCP (Bandarra & Klepper, 2026). These documents were supplied as local files.

Table 14 summarizes the Naive tool set using the categories in Table 5.

Table 14: **Naive tool set statistics.** A tool may use both declarative and imperative mechanisms, but is assigned a single primary functional role.

Application	Total	Declarative	Imperative	Read-only	Interaction	Navigation
Reddit	20	0	20	11	3	6
GitLab	26	0	26	18	2	6
Shopping	22	0	22	11	4	7
Shopping Admin	18	0	18	9	5	4
Map	20	0	20	14	2	4
Wikipedia	20	0	20	13	1	6
Overall	126	0	126	76	17	33

A.8.2 Task-Paired Changes Relative to the Original Agent

Tables 1 and 16 compare the main PACT tool set and Naive tool set, respectively, with the same Original agent baseline for AgentOccam with Qwen-3.7-Plus. Both use the categories defined in Table 15. Panel (a) counts baseline failures that become successes (Gains) and baseline successes that become failures (Regressions). Panels (b) and (c) count decreases and increases in total tokens and executed steps, respectively; unchanged tasks are omitted. Each changed task receives one category per metric; row totals therefore count changed tasks, and a task may appear in multiple panels.

In panel (b), the token difference is decomposed into the action-count difference multiplied by the mean tokens per action and the tokens-per-action difference multiplied by the mean action count across the paired runs. The larger absolute component determines the category; missing or zero action counts fall under Other. In panel (c), Longer success and Earlier failure identify changes associated with opposite task outcomes before the remaining tasks are classified by the larger net contribution of tool use or predefined actions in the direction of the step change. Tied contributions fall under Other.

Table 15: **Attribution categories for paired task changes.** The same categories apply to both tool sets and both directions of change.

Category	Definition
Success changes	
Action efficiency	Combining operations or avoiding repeated actions can enable completion within the action budget; inefficient execution can exhaust that budget before completion.
Action selection	Selecting an appropriate action or target advances the task, whereas choosing the wrong operation, object, or navigation path can prevent completion.
Parameter handling	Correct parameters and entity matches help satisfy task constraints; incorrect values or interpretations can produce an unintended result.
Information access	Access to relevant records and fields supports a complete answer; missing information or an inappropriate retrieval scope can lead to an incorrect answer.
State handling	Successful handling of state transitions during authentication, execution, or recovery enables progress; failure to establish the required state can block completion. For example, recovery can restore a usable state after an incorrect action.
Prior-knowledge	After retrieval fails, prior knowledge or estimates may yield an accepted answer, but inaccurate or insufficiently specific answers can fail evaluation.
Other	Other recorded factors include unrecoverable errors and safety refusals that prevent completion, as well as grading differences on similar results.
Token changes	
Action count	Fewer or more actions reduce or increase total tokens; the action-count component is larger than the tokens-per-action component.
Token / action	Lower or higher token use per action reduces or increases total tokens; this component is larger than the action-count component.
Other	Token usage differs, but zero or unavailable action counts prevent attribution to either component.
Step changes	
Tool use	The net change in tool-use steps contributes most in the direction of the total step change.
Predefined actions	The combined net change in predefined actions, including clicks, text entry, and navigation, contributes most in the direction of the total step change.
Longer success	The tool run completes the task using more actions than the failed baseline, increasing steps while improving success.
Earlier failure	The tool run fails after fewer actions than the successful baseline, reducing steps without improving task completion.
Other	Steps change, but the available action breakdown does not identify a unique dominant action type.

Table 16: **Naive tool set versus Original agent: paired-change attribution.** AgentOccam with Qwen-3.7-Plus on 812 WebArena tasks; the tool condition uses Full Page + Full PACT. Counts follow Table 15.

(a) Success changes

Change	Action efficiency	Action selection	Parameter handling	Information access	State handling	Prior-knowledge	Other
Regressions	5	4	1	17	6	6	12
Gains	14	12	1	9	1	3	19

(b) Token changes

Change	Action count	Token / action	Other
Increases	178	295	13
Decreases	203	119	4

(c) Step changes

Change	Tool use	Predefined actions	Longer success	Earlier failure	Other
Increases	182	65	22	0	7
Decreases	0	314	0	17	0

Across both tool sets, token reductions are more often associated with fewer actions, and step reductions with fewer predefined actions. The Naive tool set has slightly more gains than regressions, with information access accounting for the largest named regression category, illustrating why resource savings must be considered alongside task outcomes.

A.9 Screenshot Ablation

The no-screenshot conditions remove image inputs from ColorBrowserAgent's action selection and progress summarization while retaining textual page observations, action history, and PACT tool information. Predefined page interactions target element identifiers in the accessibility tree rather than screenshot coordinates, so they remain available without images. Under Full Page, textual page observations are supplied by default; under Lightweight Page, they are available on request.

Table 17: **Screenshot-ablation metrics.** ColorBrowserAgent is evaluated without screenshots on the same 812 WebArena tasks. Matched conditions with screenshots are reported in Table 3. SR subscripts compare against the Original agent with screenshots.

Condition	SR (%)	Tokens		Output tokens		Cache hit (%)		Steps		Tool executions	
		all	succ.	all	succ.	all	succ.	all	succ.	all	succ.
Qwen-3.7-Plus											
Full Page + Full PACT (no screenshot)	65.6$_{+14.9\%}$	147.9k	135.6k	11.5k	9.6k	27.0	27.4	6.0	5.4	2.4	2.2
Lightweight Page + Full PACT (no screenshot)	64.8$_{+14.0\%}$	122.1k	106.7k	12.4k	10.7k	46.3	49.1	6.8	6.2	3.7	3.4
Gemini-3.7-Flash											
Full Page + Full PACT (no screenshot)	57.4$_{-6.2\%}$	99.1k	92.4k	1.8k	1.6k	9.3	8.5	4.5	4.2	1.9	1.8
Lightweight Page + Full PACT (no screenshot)	55.3$_{-8.3\%}$	87.9k	88.7k	3.5k	3.5k	5.4	5.8	6.5	6.2	3.0	3.0

A.10 Complete Model-Size Results

Table 18: **Complete model-size metrics.** AgentOccam is evaluated across models in Qwen-3.5 family over the 812 WebArena tasks.

Model	Condition	SR (%)	Tokens		Output tokens		Steps		Tool executions	
			all	succ.	all	succ.	all	succ.	all	succ.
Qwen3.5-122B-A10B	Original agent	35.5	234.0k	97.2k	18.8k	9.0k	18.8	9.4	N/A	N/A
	Full Page + Full PACT	49.0 +13.5%	194.7k	75.2k	13.6k	5.7k	12.1	6.6	3.9	2.8
Qwen3.5-27B	Original agent	38.9	208.5k	103.4k	15.7k	7.5k	14.3	8.7	N/A	N/A
	Full Page + Full PACT	52.3 +13.4%	135.8k	62.6k	8.6k	3.6k	9.5	5.7	3.6	2.6
Qwen3.5-9B	Original agent	34.1	179.0k	89.0k	10.7k	5.8k	12.3	7.5	N/A	N/A
	Full Page + Full PACT	40.8 +6.7%	100.2k	74.2k	5.6k	4.2k	7.5	6.2	2.4	1.8
Qwen3.5-4B	Original agent	32.9	128.5k	81.8k	10.2k	6.7k	8.6	6.4	N/A	N/A
	Full Page + Full PACT	44.5 +11.6%	158.9k	134.8k	11.6k	9.2k	8.3	7.1	2.4	2.3
Qwen3.5-2B	Original agent	6.5	84.6k	68.5k	29.1k	21.6k	2.2	2.0	N/A	N/A
	Full Page + Full PACT	10.5 +3.9%	127.1k	109.4k	30.1k	17.4k	3.3	3.5	0.9	1.2

A.11 Complete Fine-Tuning Study Results

Table 19: **Complete fine-tuning study metrics.** For OpAgent and A3, base and fine-tuned checkpoints are compared on 812 WebArena tasks.

Model	Condition	SR (%)	Tokens		Output tokens		Steps		Tool executions	
			all	succ.	all	succ.	all	succ.	all	succ.
OpAgent (Guo et al., 2026)										
Qwen3-VL-32B-Thinking	Original agent	18.2	96.1k	34.0k	4.8k	2.5k	13.6	7.3	N/A	N/A
	Full Page + Full PACT	29.8 +11.6%	82.7k	35.9k	4.4k	2.5k	11.4	6.8	2.6	2.6
OpAgent-32B	Original agent	22.5	48.9k	28.8k	2.7k	1.7k	10.7	7.6	N/A	N/A
	Full Page + Full PACT	28.9 +6.4%	58.3k	32.9k	2.9k	1.9k	10.4	6.8	2.2	2.5
A3 (Lù & Reddy, 2026)										
Qwen3.5-9B	Original agent	36.3	165.5k	92.2k	5.0k	2.2k	12.6	7.5	N/A	N/A
	Full Page + Full PACT	41.1 +4.8%	148.8k	94.0k	3.9k	2.1k	11.6	7.3	2.3	1.2
A3-Qwen3.5-9B	Original agent	47.2	201.1k	100.8k	10.9k	4.8k	15.1	8.6	N/A	N/A
	Full Page + Full PACT	47.7 +0.5%	218.5k	116.1k	12.9k	5.7k	14.5	8.1	1.3	0.5

A.12 Standalone Mock Chinese Consumer-Site Extension

Kimi-K3 is prompted to generate mock versions of Xiaohongshu, Bilibili, Weibo, and 12306. GPT-5.5 is prompted to generate their PACT tools following Section 3.2 and Appendix A.1. No user content, trademarked material, or other sensitive material was retained during generation. On these sites, AgentOccam with Qwen-3.7-Plus is evaluated under the Original agent and Full Page + Full PACT conditions on 90 tasks designed by us (20 single-site tasks per site and 10 cross-site tasks). Evaluation follows the main-study protocol.

Table 20: **Mock Chinese consumer-site extension.** AgentOccam with Qwen-3.7-Plus is evaluated over 90 standalone tasks (20 single-site tasks per site and 10 cross-site tasks) designed by us.

Model	Condition	SR (%)	Tokens		Output tokens		Cache hit (%)		Steps		Tool executions	
			all	succ.	all	succ.	all	succ.	all	succ.	all	succ.
Qwen-3.7-Plus	Original agent	41.1	75.4k	74.0k	5.9k	5.9k	14.2	13.9	6.1	6.1	N/A	N/A
	Full Page + Full PACT	51.1 +10.0%	59.2k	53.6k	3.8k	3.4k	14.6	14.6	4.6	4.2	2.6	2.1

In this extension, Full Page + Full PACT raises success by 10 percentage points and reduces all-task mean tokens by 21.5% and steps by 24.6%.