

Apex Browse: Deterministic Browser Workflow Execution with Bounded Large-Language-Model Repair

Jagpreet Singh Sasan

Independent Researcher

Corresponding email: jagjasinsa@gmail.com

Abstract

Large-language-model agents can operate web interfaces through browser automation tools, but the common stepwise pattern repeatedly sends page observations to a model and requests the next action. This adds inference latency, token consumption, and opportunities for untrusted page content to influence control decisions. This paper presents Apex Browse, an open-source Model Context Protocol server that moves routine browser execution out of the language-model loop. An agent expresses a workflow once in a schema-validated JSON domain-specific language; a session-local Playwright executor resolves accessibility-oriented targets, performs allow-listed actions, checks postconditions, and retains compact evidence. Model assistance is requested only when deterministic resolution cannot identify one safe target. The repair packet contains the failed intent and no more than five candidates, and a repair may only select from that set. In a controlled benchmark of five workloads, three execution arms, and ten repetitions per workload-arm pair, Apex Browse completed 49 of 50 agent attempts. Relative to the official Playwright MCP baseline, it reduced summed wall time by 45.9%, token consumption by 54.8%, MCP calls by 68.0%, and serialized MCP-result bytes by 74.3%. Across 49 jointly successful paired trials, the mean wall-time difference favoring Apex Browse was 22.52 seconds, with a deterministic bootstrap 95% confidence interval of 19.62–25.50 seconds. The results support deterministic-first execution for known browser workflows while exposing limits involving task diversity, model dependence, visual interfaces, and one agent-level failure.

Keywords: browser agents, deterministic execution, large language models, Model Context Protocol, Playwright, token efficiency

I. INTRODUCTION

A. Problem Context

Language-model agents increasingly interact with software through browser user interfaces rather than application-specific APIs. This approach has broad reach: a browser surface is often available even when a formal integration is absent, and users can inspect the same interface the agent manipulates. However, browser control also creates an expensive feedback loop. A conventional agent observes the page, reasons about the next action, invokes a browser tool, receives a new observation, and repeats. Even when the requested workflow is already known—navigate to a page, fill fields, click a named control, and verify a result—the model may be consulted after every deterministic state transition.

The expense is not limited to model output. Page snapshots, accessibility trees, tool schemas, and prior interaction history become input context on subsequent turns. Repeated serialization increases wall time and token consumption, while longer trajectories create more locations where the agent can make an unnecessary call, select the wrong element, or stop without verifying the intended state. Repeatedly exposing web content to a privileged model also enlarges the pathway through which indirect prompt injection can affect tool decisions. These costs are architectural: improving the language model alone does not remove the repeated observation–reasoning–action protocol.

A different division of labor is possible. The model is valuable when translating an open-ended user request into intent and when interpreting an unexpected semantic change. It is not necessary for executing every already-specified interaction. Browser engines and automation libraries are designed to perform such interactions deterministically. Playwright, for example, provides role-based locators, strict uniqueness behavior, actionability checks, and automatic waiting [7], [8]. The design question is therefore not whether to use an LLM or deterministic automation, but where the boundary between them should be placed.

B. Objective and Research Questions

This study investigates a deterministic-first architecture for agent-driven browsing. Apex Browse is implemented as a Model Context Protocol (MCP) server over Playwright. The model produces a complete declarative browser program; Apex Browse owns browser state and executes the program locally. A model is brought back into the loop only when a target is missing or ambiguous, and even then it receives a bounded candidate packet rather than an unrestricted page snapshot.

The work addresses three research questions:

1. **RQ1—Efficiency:** Can deterministic-first execution reduce end-to-end time, model tokens, MCP calls, and serialized MCP results compared with a stepwise browser MCP baseline?
2. **RQ2—Correctness:** Can these reductions be achieved while preserving independently observed task completion rather than relying on the agent's self-report?
3. **RQ3—Control:** Can selector drift and ambiguity be handled through a constrained repair mechanism that prevents the model from rewriting the remaining workflow or choosing an unobserved target?

C. Contributions

This paper makes four contributions. First, it defines a plan-once, execute-locally architecture that treats language-model inference as an exception path rather than the default browser-control loop. Second, it presents a compact JSON domain-specific language (DSL), accessibility-oriented resolver, revisioned semantic index, evidence receipts, and bounded repair protocol. Third, it describes safety invariants that exclude arbitrary page code, reject ambiguous mutations, label page-derived content as untrusted, and require an out-of-band host decision for high-impact actions. Fourth, it reports a reproducible controlled benchmark with an exact server-side oracle, counterbalanced arm ordering, append-only raw measurements, complete agent transcripts, explicit sanitization, and paired bootstrap analysis. The implementation and benchmark artifacts are publicly available [13]–[16].

The claimed scope is deliberately narrower than general autonomous web navigation. Apex Browse is an execution runtime for workflows whose intent can be expressed before execution. It is designed to preserve a model fallback for unexpected semantics, not to eliminate models from all browsing tasks. Open-ended research, visually grounded decisions, and tasks whose goals evolve after each observation remain outside the current prototype's strongest use case.

II. LITERATURE REVIEW

A. Interleaved Reasoning and Acting

ReAct established a widely used pattern in which a language model interleaves reasoning, actions, and observations [1]. This pattern is valuable in uncertain environments because new evidence can alter the plan. Web interaction is a natural application: each click changes a partially observable state, and a model can use the resulting observation to decide what follows. The same generality, however, means that a model may be invoked for transitions that are already predictable. Apex Browse does not dispute the value of interleaved reasoning for exploration. It proposes a conditional use of that pattern: compile stable intent into a program, execute deterministic transitions without additional inference, and re-enter a reasoning step only on a bounded exception.

This distinction can be framed as one between *planning uncertainty* and *execution certainty*. When a goal is open-ended, the next action may depend on newly discovered content and interleaving is appropriate. When the goal already names the required fields, controls, and postcondition, the principal uncertainty is whether the current interface still exposes those semantics. Apex Browse therefore concentrates model attention on semantic mismatch rather than routine execution.

B. Generalist Web Agents and Observation Size

Mind2Web introduced more than 2,000 open-ended tasks across 137 websites and 31 domains, emphasizing the diversity required of generalist web agents [2]. It also identified raw web representations as too large for direct model consumption and demonstrated the value of filtering candidate elements before LLM processing. Apex Browse follows a related efficiency principle but applies it at runtime: the normal path sends no repeated page representation, snapshot requests are capped, and failed resolution exposes at most five candidates.

WebArena advanced functional evaluation by providing realistic, self-hosted websites and task-specific evaluators [3]. Its results showed a substantial gap between agent and human task completion, reinforcing that plausible text responses are not adequate evidence of successful UI action. Apex Browse adopts the same broad measurement principle in a smaller controlled setting: success is established by server-side state, not by the agent saying that it finished.

WebVoyager demonstrated the value of combining screenshots and textual observations for agents operating public websites [4]. Its evaluation covered tasks on 15 real sites and showed that multimodal input can outperform text-only operation. Apex Browse currently makes the opposite experimental trade-off: it targets semantic workflows and omits image responses in both compared MCP arms. Consequently, its results should not be generalized to tasks where layout, imagery, charts, maps, or visual similarity determine the correct action.

WorkArena and the later BrowserGym ecosystem focus on reproducible evaluation of browser agents over knowledge-work and other web tasks [5], [6]. BrowserGym explicitly distinguishes unrestricted executable actions from constrained high-level primitives, noting the flexibility and security trade-off. Apex Browse occupies the constrained end of that design space. Its action vocabulary is intentionally small, and model output is parsed as data rather than evaluated as browser code.

C. Semantic Browser Automation

The World Wide Web Consortium's WAI-ARIA specification defines roles, states, properties, and accessible naming semantics for user-interface elements [7]. Playwright exposes these semantics through role and label locators and recommends user-facing attributes for resilient automation [8]. It also waits for relevant conditions such as visibility, stability, event reception, enablement, and editability before performing supported actions [9]. These mechanisms provide a deterministic substrate that an agent runtime can reuse instead of reconstructing browser behavior through natural-language reasoning.

Apex Browse targets controls by accessible role and name, with optional scope and explicitly supplied aliases. The approach is more generic than site-specific CSS or XPath selectors because it describes user-facing intent, yet less permissive than allowing the model to issue arbitrary Playwright code. It depends on sites exposing meaningful semantics; poorly labeled interfaces remain a limitation.

D. Model Context Protocol and Browser Tooling

MCP defines a client–host–server architecture in which servers expose tools with names, descriptions, and JSON schemas [10]. This makes browser capabilities discoverable across compatible agent hosts. Microsoft's Playwright MCP server provides a general browser interface and serves as the official baseline in this study [11]. A general interface is suitable for exploratory browsing, but repeated tool calls and page observations can be costly for known workflows. Apex Browse uses the same interoperability layer while changing the granularity of the primary tool call: `apex_browse_run` accepts an entire validated program.

E. Security Motivation

Web pages are external inputs and can contain instructions directed at an agent rather than information relevant to the user's task. OWASP identifies prompt injection as a major LLM application risk and recommends structured output validation, least privilege, segregation of external content, and human approval for high-risk actions [12]. MCP likewise treats tool metadata and results with caution and places responsibility for consent and authorization on implementations [10]. Apex Browse cannot prove that prompt injection is solved. Instead, it reduces exposure and limits consequences: routine actions are derived from a prevalidated program, page-derived text is marked untrusted, arbitrary code is unavailable, repairs are restricted to observed candidates, and sensitive actions require host approval.

F. Research Gap

Existing work largely asks how to make an agent perceive, reason, and act more effectively across diverse web tasks. Apex Browse asks a complementary systems question: once an agent has formed a usable plan, how much of the remaining browser interaction can be removed from the agent loop? The novel element is not a new foundation model or locator algorithm. It is an execution boundary that combines a complete declarative program, session-local semantic state, deterministic postconditions, and a narrowly authorized return to model reasoning.

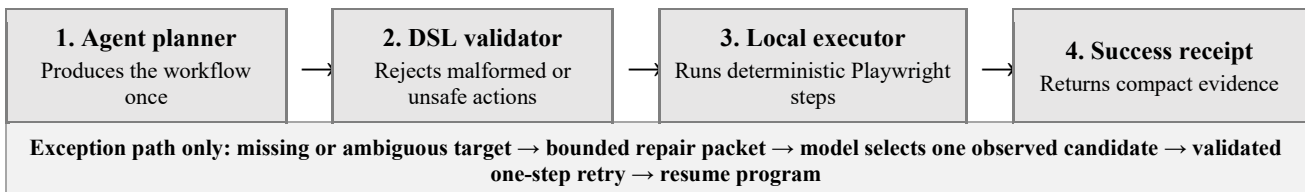
III. METHODOLOGY

A. System Architecture

The architecture contains five logical components: an external planner, a DSL validator, a session-local executor, a semantic resolver, and a bounded repair interface. The planner may be a host agent or a specialized skill. It translates the user's request into a complete JSON program. The validator rejects malformed or out-of-policy actions. A single executor owns one Playwright page and runs validated steps in sequence. The resolver maps semantic targets to visible controls. If one safe target cannot be found, execution pauses and the MCP server returns a repair packet. Detailed evidence remains in local memory and is retrieved only on demand.

The nominal flow is:

FIGURE 1. DETERMINISTIC-FIRST EXECUTION AND BOUNDED REPAIR PATH



This arrangement separates two state machines. The browser state machine is held by Apex Browse and advances after local actions. The conversation state remains with the host agent. The host does not need a new page snapshot after every successful browser transition, and a subagent or repair skill does not receive direct ownership of the page.

B. Domain-Specific Language

A program is an ordered sequence:

$$P = \langle a_1, a_2, \dots, a_n \rangle, \text{ where } 1 \leq n \leq 50. \quad (1)$$

The implemented action set is navigate, click, fill, select, check, press, and expect. Mutating actions reference a target containing an accessible role, a non-empty accessible name, up to eight optional aliases, and an optional recursively defined scope. Supported roles are button, checkbox, combobox, dialog, link, radio, searchbox, and textbox. An expectation checks visible text or a URL substring. The language deliberately has no operation for arbitrary JavaScript, browser evaluation, shell execution, XPath, CSS selectors, or unconstrained Playwright expressions.

A representative program is:

```

{
  "steps": [
    { "op": "navigate", "url": "https://portal.example.test/request" },
    {
      "op": "fill",
      "target": { "role": "textbox", "name": "Contact email" },
      "value": "researcher@example.test"
    },
    {
      "op": "click",
      "target": { "role": "button", "name": "Submit request" }
    },
    { "op": "expect", "text": "Request received" }
  ]
}
  
```

Parsing is performed before execution using a discriminated schema. Thus, language-model output crosses a validation boundary before it can affect the page. The complete program also makes the intended sequence inspectable and loggable before the first mutating step.

C. Semantic Index and Resolution

For each page revision, the executor builds and caches a private control index. Each entry contains a revision-local identifier, accessible role, accessible name, disabled state, and checked state where applicable. Hidden controls and unnamed controls are excluded. Names are derived from `aria-label`, `aria-labelledby`, associated labels, inner text, or input hint text in precedence order. A public snapshot is capped at 24 controls and 800 visible-text characters; local search returns a caller-bounded candidate list without serializing the entire page.

For target t and indexed page state S_r at revision r , the deterministic resolution function $R(S_r, t)$ is defined as follows:

Resolution function (2)

- $\text{unique}(e)$ when $|E(t)| = 1$;
- $\text{ambiguous}(E(t))$ when $|E(t)| > 1$;
- $\text{unique}(e)$ when $|E(t)| = 0$ and $|N(t)| = 1$;
- $\text{ambiguous}(N(t))$ when $|N(t)| > 1$;
- $\text{missing}(C(t))$ otherwise.

Here, $E(t)$ is the set of exact role-and-name matches, $N(t)$ is the set matching the normalized requested name or an explicit alias, and $C(t)$ is a capped same-role candidate set. Normalization lowercases text, converts punctuation runs to spaces, trims, and collapses whitespace. The implementation does not perform open-ended fuzzy matching. This conservative choice protects the mutation invariant:

$$\text{mutate}(t) \Rightarrow |R(S_r, t)| = 1. \quad (3)$$

Scope is resolved recursively, enabling a target such as a Save button inside a named dialog. Selection controls accept either an exact option value or a unique normalized visible label. After a mutating action, the revision advances and the cached index is invalidated.

D. Bounded Repair

When resolution returns `missing` or `ambiguous`, execution pauses before mutation. The repair packet contains a run identifier, failed step number, original action, reason, page revision, and at most five controls. It is explicitly marked `untrusted`. A repair request is valid only if its replacement role and name exactly match one candidate in that packet. The executor replaces the target of the paused action, clears aliases, retries that action, and then resumes the original remaining program.

The repair path therefore cannot change the action type, field value, earlier receipts, subsequent steps, or postcondition. It also cannot introduce a target that the executor did not expose. In authorization terms, the model receives a finite choice set rather than a new program-writing capability. An ambiguous mutating target remains non-mutating until the host disambiguates it.

Failed postconditions are handled differently. They return a failure receipt and cannot be repaired by modifying the assertion. This prevents an agent from converting unsuccessful execution into apparent success by weakening the acceptance criterion.

E. Safety and Evidence

Four controls constrain execution. First, the DSL is an allow-list and never evaluates generated browser code. Second, every snapshot, search result, and repair packet identifies page-derived content as `untrusted`. Third, ambiguity is fail-closed for mutation. Fourth, target names associated with high-impact operations require both `confirm: true` in the program and a separate `approveHighImpact` callback supplied by the host. The MCP caller cannot satisfy the second condition by modifying its own tool arguments.

Every successful step produces a receipt containing the step index, operation, evidence identifier, and, when applicable, the resolved role, name, and resolution path. Evidence is retained locally under memory identifiers. The normal MCP response returns the compact identifiers and metrics rather than automatically serializing all retained details.

These controls reduce attack surface but do not establish a complete security boundary. A user-approved program can still encode harmful intent; target-name heuristics cannot classify every consequential action; and the host callback is only as trustworthy as its implementation. The security contribution is constrained execution and auditable authorization, not a claim of immunity from malicious sites or compromised hosts.

F. MCP Surface and Implementation

The TypeScript prototype exposes six tools: navigation, complete-program execution, bounded snapshot retrieval, local semantic search, constrained repair, and evidence retrieval. Playwright supplies Chromium control and actionability behavior. The MCP server communicates over standard input/output and maintains the session in process memory. Two small skills define the planner and repair contracts: the planner sends a complete DSL program, while the repair skill receives only the repair packet.

The implementation at the evaluated revision is approximately a prototype rather than a production browser platform. It has one page per session, in-memory state, no visual model, a limited role vocabulary, and no persistent authentication manager. These choices keep the experimental mechanism inspectable and make the measured difference attributable primarily to interaction granularity rather than a large supporting platform.

G. Experimental Design

The benchmark compares three arms:

1. **Apex Browse:** the compiled stdio MCP server in the repository;
2. **Official Playwright MCP:** `@playwright/mcp@0.0.79`, using headless isolated Chromium, default full semantic snapshots, and omitted image responses;

3. **Native Playwright:** fixed direct Playwright code, included only as a non-agent lower-bound context.

Both agent arms use Codex CLI with gpt-5.6-luna, low reasoning effort, an ephemeral thread, and user configuration disabled. Each receives identical task wording and is prohibited from using shell, file, web-search, or non-browser tools. The native arm receives no model and is not interpreted as an agent competitor.

Five local workloads exercise distinct interaction patterns:

- **Profile:** fill two names and an email, select a role, check terms, submit, and verify success;
- **Profile drift:** complete the same semantic task after structural changes to the form;
- **Dialog:** open a dynamic dialog, fill scoped controls, select an option, check a preference, and save;
- **Renamed control:** submit a message where the expected action is expressed against a semantically changed control;
- **Catalog:** enter a product query, search, open the matching product, and verify the details state.

There are ten repetitions for each workload–arm combination, yielding 150 attempts: 50 Apex Browse, 50 official MCP, and 50 native. Arm order rotates cyclically across workload and trial pairs to reduce ordering and warm-system bias. Each attempt starts fresh Codex, MCP, and isolated browser processes. Dependencies and browser binaries are warmed before measurement, but agent, MCP, and browser startup remain inside elapsed time.

H. Independent Oracle and Recorded Measures

The controlled website records the state-changing payload on the server. Before every attempt, this state is reset. A trial succeeds only when the recorded object exactly equals the workload's expected object. Neither a DONE message nor a zero process exit code is sufficient. This avoids relying on model self-evaluation and distinguishes task completion from confident narration.

The runner appends one JSON object per trial containing expected and observed state, elapsed time, timeout and exit status, tool names, tool errors, serialized MCP-result bytes, final agent response, and all token counters exposed by the host. Complete JSON event transcripts are retained for agent arms. The benchmark also records Apex Browse's self-reported local execution time, action count, and repair count when available.

For a metric x , aggregate reporting includes the mean, sample standard deviation, median, and 95th percentile. Successful-duration statistics exclude failed attempts, while all-attempt time and success rate are retained. Agent arms are paired by workload and repetition. For jointly successful pairs, differences are defined as:

$$d_i = x_{\text{official},i} - x_{\text{Apex Browse},i} \quad (4)$$

A deterministic 10,000-resample bootstrap estimates the 95% confidence interval for the mean paired difference, following the general bootstrap method described by Efron and Tibshirani [17]. Positive differences favor Apex Browse. The analysis does not treat the native arm as part of this paired agent comparison.

I. Reproducibility and Privacy

The benchmark protocol, runner, controlled site, analyzer, verifier, raw JSONL, summary, report, and 100 agent transcripts are published at a commit-pinned repository revision [13]–[16]. The principal reproduction sequence is:

```
npm.cmd install
npx.cmd playwright install chromium
npm.cmd run benchmark:run -- --trials 10
npm.cmd run benchmark:analyze
npm.cmd run benchmark:verify
```

An append-only resume mechanism validates frozen metadata and skips only completed workload–trial–arm keys. Before publication, a separate sanitizer removes hardware, operating-system, timezone, user-directory, absolute repository path, dynamic local port, and conversation-thread identifiers. Sanitization occurs after measurement and does not recompute timing, token, call, or measured byte values.

IV. RESULTS

A. Aggregate Outcomes

Table I reports the principal aggregate results. Apex Browse succeeded in 49 of 50 attempts (98%), while official Playwright MCP and native Playwright succeeded in 50 of 50. Apex Browse accumulated 1,347,415.04 ms of agent-arm wall time versus 2,490,694.41 ms for the official MCP arm, a reduction of 45.9%. Its median successful time was 27,485.76 ms, compared with 49,765.97 ms. The corresponding 95th percentiles were 37,518.49 ms and 62,359.55 ms.

TABLE I. AGGREGATE BENCHMARK RESULTS

Arm	Independent success	Total time (min)	Median successful time (ms)	p95 successful time (ms)	Total tokens	Median MCP calls	Total MCP-result bytes
Apex Browse	49/50 (98%)	22.46	27,485.76	37,518.49	4,393,321	2	59,006
Official Playwright MCP	50/50 (100%)	41.51	49,765.97	62,359.55	9,726,675	7	229,641
Native Playwright	50/50 (100%)	0.22	264.04	333.20	0	0	0

Apex Browse consumed 4,393,321 total tokens versus 9,726,675, a 54.8% reduction. It made 115 MCP calls versus 359, a 68.0% reduction, and received 59,006 serialized MCP-result bytes versus 229,641, a 74.3% reduction. The official arm recorded 25 failed MCP call events, all recoverable within successful overall attempts; Apex Browse recorded none. Estimated model cost under the frozen pricing rule was USD 1.367294 for Apex Browse and USD 2.113408 for official MCP, a 35.3% difference. Cost is secondary to the raw counters because pricing is time-dependent.

B. Workload-Level Results

Table II shows that the efficiency pattern appears in every workload, though the size varies. The profile-drift task produced the largest median wall-time ratio: 19.34 seconds for Apex Browse versus 53.57 seconds for official MCP. In this case, explicit semantic aliases and local execution allowed a median of one MCP call. The dialog workload required dynamic discovery and scope, resulting in a median of two Apex Browse calls versus nine official calls. The catalog workload was the most demanding Apex Browse case, with a median of four calls and the only failure.

TABLE II. RESULTS BY WORKLOAD

Workload	Arm	Success	Median successful time (ms)	Median total tokens	Median MCP calls
Profile	Apex Browse	10/10	27,546.48	81,418.5	2
Profile	Official MCP	10/10	46,253.88	189,168.5	6
Profile drift	Apex Browse	10/10	19,341.01	57,415.0	1
Profile drift	Official MCP	10/10	53,571.15	187,918.5	9
Dialog	Apex Browse	10/10	30,771.26	85,297.0	2
Dialog	Official MCP	10/10	55,345.66	199,541.0	9
Renamed control	Apex Browse	10/10	25,962.42	80,984.5	2
Renamed control	Official MCP	10/10	43,361.81	200,451.0	6
Catalog	Apex Browse	9/10	30,405.36	126,057.5	4
Catalog	Official MCP	10/10	44,214.96	176,583.0	7

The workload data support the proposed mechanism rather than only an aggregate effect. When the complete task fit a known semantic program, several local steps were compressed into one or two tool calls. Workloads involving dynamic state or repair required additional calls but remained below the corresponding official medians.

C. Paired Differences

Forty-nine workload-trial pairs were jointly successful. For those pairs, official MCP minus Apex Browse wall time had a mean of 22,521.06 ms, a sample standard deviation of 10,505.07 ms, and a bootstrap 95% confidence interval of 19,620.91–25,497.17 ms. The median official-to-Apex-Browse time ratio was 1.804.

The mean paired token difference was 105,561.22 tokens with a sample standard deviation of 62,747.16 and a bootstrap 95% confidence interval of 88,214.48–122,439.24 tokens. The median official-to-Apex-Browse token ratio was 2.484. One paired trial had a negative token difference, so the result is not a claim that Apex Browse dominates on every individual run. The aggregate and interval estimates nevertheless show a consistent average reduction in this controlled workload set.

D. Correctness and Failure Analysis

All successful classifications were verified by exact server-side state. Apex Browse matched the oracle in 49 attempts and official MCP in 50. The sole Apex Browse failure occurred in catalog trial 4. The transcript records that the agent returned FAILED without calling any MCP tool. It was therefore classified as `agent_declined_without_tool`, not as a browser runtime, resolver, repair, or oracle failure. Retaining this trial in all-attempt time, token totals, and success rate prevents selective reporting.

The 98% versus 100% success difference is important despite the failure's location outside the executor. End-to-end agent systems include model decisions about whether to invoke tools. A deterministic runtime cannot optimize a workflow it is never asked to execute. Future work should therefore examine planner adherence and tool-selection reliability independently from browser execution.

E. Local Execution Cost

For 49 Apex Browse attempts reporting local metrics, median DSL execution time was 252.87 ms and the 95th percentile was 2,539.73 ms. Median end-to-end successful time was 27,485.76 ms. The large gap indicates that model, host, MCP, and browser-process orchestration—not local locator execution—dominated measured latency. Native Playwright's 264.04 ms median provides a consistent lower-bound context, although it uses fixed code and no agent startup. These observations support batching known actions into a local run: reducing agent interactions addresses the dominant cost without requiring Playwright itself to become materially faster.

V. DISCUSSION

A. Interpretation of the Efficiency Gains

The primary result is a change in control granularity. Official Playwright MCP exposes capable browser operations, but the agent generally chooses and verifies actions across multiple calls. Apex Browse asks the model to externalize a complete intent program and then delegates the predictable portion of the trajectory to deterministic code. Fewer round trips also mean fewer repeated snapshots and smaller accumulated conversational context. The measured reductions in tokens, calls, and result bytes are mutually consistent with this mechanism.

The result should not be interpreted as evidence that a DSL is inherently more intelligent than an agentic browser. It is evidence that model intelligence need not be spent on every state transition. Apex Browse relies on the host model for task interpretation and on Playwright for browser mechanics. Its contribution lies in coordinating those capabilities around a narrow exception boundary.

B. Correctness–Efficiency Trade-off

Apex Browse reduced cost while losing one end-to-end attempt. The observed failure was an agent refusal to invoke any tool, so it does not reveal a local execution defect. It nevertheless prevents a simple claim of equal reliability. For deployment, optimization should be evaluated against both per-run cost and the operational cost of retries or unresolved tasks. A 54.8% token reduction may justify retrying rare planner-level failures in some settings, but high-assurance workflows may prioritize the baseline's observed 100% completion until larger studies characterize failure rates.

Postconditions are central to this balance. Batching actions without verification would be fast but unsafe. Apex Browse embeds expect steps and treats a failed postcondition as a terminal failure rather than a repairable statement. This design keeps the acceptance criterion fixed while allowing only target-level adaptation.

C. Genericity Without Site-Specific Scripts

The architecture is intended to be generic across semantically accessible websites. Programs target roles and accessible names rather than DOM paths, and the runtime builds its control index at execution time. The benchmark includes structural drift, a dynamic dialog, a renamed control, and a multi-stage catalog rather than only one static form. These cases demonstrate limited cross-structure flexibility.

They do not establish whole-web generality. Real sites include iframes, shadow DOM, virtualized lists, custom canvases, anti-automation mechanisms, authentication transitions, file uploads, rich text editors, and visually meaningful controls with poor accessibility metadata. Some tasks require choosing among products, interpreting images, or changing the plan after reading content. In those cases, more frequent observation or a multimodal agent may be necessary. A production system should route tasks between deterministic execution, bounded repair, and full exploratory control according to uncertainty rather than forcing every task through one mode.

D. Safety Implications

The design reduces two forms of authority. It reduces *information authority* by withholding unrestricted page content on the normal path, and it reduces *action authority* by limiting the model to an allow-listed program and candidate-bound repair. These constraints align with OWASP recommendations to validate structured outputs, use least privilege, separate external content, and require human approval for high-risk actions [12].

However, the current system's untrusted marker is metadata, not a formal information-flow guarantee. A host could ignore it, retrieve a full snapshot, and expose that snapshot to a privileged model. Likewise, a name-based high-impact classifier cannot replace policy based on destination, user identity, data sensitivity, monetary value, or legal consequence. Stronger deployments should add origin policies, credential isolation, per-tool grants, transaction previews, durable audit logs, and host-enforced approval UI.

E. Threats to Validity

Internal validity. Processes were freshly created and arm order was rotated, but the operating system still scheduled trials and shared machine resources. Package and browser caches were warm, while startup remained measured. The model service may have variable load. Ten repetitions per workload provide limited precision for tail behavior. The same author developed and evaluated the system, creating potential implementation and analysis bias; public code, exact-oracle state, raw records, transcripts, and a verifier reduce but do not eliminate this risk.

Construct validity. Tokens, tool calls, serialized result bytes, and wall time measure interaction cost, but they do not capture all operational concerns. Estimated monetary cost depends on a frozen price schedule. Tool errors include recoverable validation failures and are not equivalent to failed tasks. The benchmark's exact oracle is strong for the selected forms but does not measure subjective quality, semantic partial credit, or the appropriateness of actions that happen to produce the expected payload.

External validity. The controlled website removes internet variability and makes server-side correctness observable, but five workloads cannot represent the web. The study uses one language model, one reasoning setting, one browser family, one official MCP version, and one run period. Images were omitted from both MCP arms, which creates a fair semantic comparison but excludes a capability important to visual agents. Results may change with other models, prompting policies, networked websites, longer workflows, or later implementations.

Statistical conclusion validity. The paired confidence intervals quantify uncertainty over the observed jointly successful pairs, not over all possible sites or models. Excluding the unmatched failed Apex Browse attempt from paired differences can favor the paired comparison; the paper therefore reports all-attempt success and aggregate time alongside successful-pair statistics. No hypothesis test is offered for the two-percentage-point success difference because the sample is too small to support a useful reliability conclusion.

F. Engineering Implications and Future Work

The findings suggest an adaptive browser stack with three modes. A known workflow should use one complete local program. A semantic mismatch should use bounded repair. A genuinely open-ended task should escalate to richer snapshots, multimodal reasoning, or conventional stepwise control. This routing policy preserves generality while making deterministic execution the inexpensive default when uncertainty is low.

Priority engineering work includes iframe and shadow-DOM traversal, multi-page sessions, downloads and uploads, origin allow-lists, persistent but isolated authentication, stronger semantic-name computation, and structured transaction previews. Evaluation should expand to public benchmark suites such as WebArena and WorkArena, multiple models, adversarial pages, accessibility-poor interfaces, and ablations that separately vary program batching, snapshot caps, aliases, and repair limits. A larger study should report retry-adjusted cost, repair precision, wrong-action rate, and failure probability with confidence intervals.

VI. CONCLUSION

Apex Browse demonstrates that an LLM need not remain in the loop for every browser action after it has expressed a complete workflow. A strict DSL, session-local Playwright executor, accessibility-oriented resolver, explicit postconditions, compact evidence, and candidate-bound repair can shift routine interaction to deterministic code while preserving a controlled path back to model reasoning.

In the reported 150-attempt controlled experiment, Apex Browse completed 49 of 50 agent tasks and substantially reduced wall time, tokens, MCP calls, and serialized result bytes relative to official Playwright MCP. The strongest empirical conclusion is limited but useful: for the tested semantic workflows under the pinned configuration, deterministic-first execution delivered materially lower interaction cost with one observed planner-level failure. The benchmark does not establish universal web-agent superiority, visual competence, or production-grade security.

The broader design principle is to reserve probabilistic reasoning for uncertainty and use deterministic systems for execution that can be fully specified and verified. Browser agents need both capabilities. Apex Browse places the boundary at failed semantic resolution and makes that boundary explicit, inspectable, and measurable.

CODE AND DATA AVAILABILITY

The Apex Browse source code is publicly available in the project repository [13]. The evaluated implementation is pinned to commit 3e1ce6b3d2f8fc8a154c3c3ab4e136e6645d84ad. The frozen benchmark methodology [14], analyzed report [15], append-only raw data [16], machine-readable summary, verifier, and sanitized agent transcripts are retained in the same repository. The repository includes commands for build, test, benchmark execution, resume, sanitization, analysis, and verification.

ETHICS, PRIVACY, AND CONFLICT OF INTEREST

The experiment used a locally controlled website and synthetic form values. It involved no human participants, private accounts, or production-site transactions. Published diagnostic artifacts were sanitized to remove machine-specific and conversation identifiers without altering measured numeric counters. The author is the developer of Apex Browse and therefore has a direct intellectual interest in the evaluated system. This conflict is disclosed; independent replication is encouraged through the public implementation, raw records, complete transcripts, exact-oracle fixture, and automated verifier.

REFERENCES

- [1] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proc. International Conference on Learning Representations*, 2023. [Online]. Available: <https://arxiv.org/abs/2210.03629>
- [2] X. Deng, Y. Gu, B. Zheng, S. Chen, S. Stevens, B. Wang, H. Sun, and Y. Su, "Mind2Web: Towards a generalist agent for the web," in *Advances in Neural Information Processing Systems*, vol. 36, 2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/hash/5950bf290a1570ea401bf9882128160-Abstract-Datasets_and_Benchmarks.html
- [3] S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, U. Alon, and G. Neubig, "WebArena: A realistic web environment for building autonomous agents," in *Proc. International Conference on Learning Representations*, 2024. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2024/hash/4410c0711e9154a7a2d26f9b3816d1ef-Abstract-Conference.html
- [4] H. He, W. Yao, K. Ma, W. Yu, Y. Dai, H. Zhang, Z. Lan, and D. Yu, "WebVoyager: Building an end-to-end web agent with large multimodal models," in *Proc. 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 6864–6890, 2024, doi: 10.18653/v1/2024.acl-long.371.
- [5] A. Drouin, M. Gasse, M. Caccia, I. H. Laradji, M. Del Verme, T. Marty, D. Vazquez, N. Chapados, and A. Lacoste, "WorkArena: How capable are web agents at solving common knowledge work tasks?" *Transactions on Machine Learning Research*, 2024. [Online]. Available: <https://openreview.net/forum?id=BRfqYrikdo>
- [6] T. Le Sellier de Chezelles *et al.*, "The BrowserGym ecosystem for web agent research," *Transactions on Machine Learning Research*, 2025. [Online]. Available: <https://openreview.net/forum?id=5298fKGmv3>
- [7] World Wide Web Consortium, "Accessible Rich Internet Applications (WAI-ARIA) 1.2," W3C Recommendation, Jun. 2023. [Online]. Available: <https://www.w3.org/TR/wai-aria/>
- [8] Microsoft, "Locators," *Playwright Documentation*. [Online]. Available: <https://playwright.dev/docs/locators>. Accessed: Aug. 9, 2026.
- [9] Microsoft, "Auto-waiting and actionability," *Playwright Documentation*. [Online]. Available: <https://playwright.dev/docs/actionability>. Accessed: Aug. 9, 2026.
- [10] Model Context Protocol Contributors, "Model Context Protocol specification," 2025. [Online]. Available: <https://modelcontextprotocol.io/specification/2025-03-26/index>
- [11] Microsoft, "Playwright MCP server," GitHub repository. [Online]. Available: <https://github.com/microsoft/playwright-mcp>. Accessed: Aug. 9, 2026.
- [12] OWASP Foundation, "LLM01:2025 Prompt injection," *OWASP Top 10 for Large Language Model Applications*, 2025. [Online]. Available: <https://genai.owasp.org/llmrisk/llm01-prompt-injection/>
- [13] J. S. Sasan, "Apex Browse: Deterministic Playwright execution with bounded LLM repair," GitHub repository, version 0.1.0, commit 3e1ce6b3d2f8fc8a154c3c3ab4e136e6645d84ad, 2026. [Online]. Available: <https://github.com/TheJagpreet/apex-browse/tree/3e1ce6b3d2f8fc8a154c3c3ab4e136e6645d84ad>
- [14] J. S. Sasan, "Apex Browse benchmark methodology," 2026. [Online]. Available: <https://github.com/TheJagpreet/apex-browse/blob/3e1ce6b3d2f8fc8a154c3c3ab4e136e6645d84ad/benchmark/methodology.md>
- [15] J. S. Sasan, "Apex Browse benchmark results," 2026. [Online]. Available: <https://github.com/TheJagpreet/apex-browse/blob/3e1ce6b3d2f8fc8a154c3c3ab4e136e6645d84ad/benchmark/results/report.md>
- [16] J. S. Sasan, "Apex Browse raw benchmark records and agent transcripts," 2026. [Online]. Available: <https://github.com/TheJagpreet/apex-browse/blob/3e1ce6b3d2f8fc8a154c3c3ab4e136e6645d84ad/benchmark/results/raw-luna-2026-08-09.jsonl>
- [17] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. New York, NY, USA: Chapman & Hall, 1993.