Solid-State Web Execution: Compiling Web Applications into Verified Operations for AI Agents

Vahagn Boudakian · Protocolis · vahagn@protocolis.co

Abstract

A web agent pays a language model to read the page at every step: first to work out what it can do there, then to find out whether its last action worked. Solid-state web execution removes both. On arrival at a page it has never seen, the Protocolis compiler turns the page's controls into typed operations; the model picks one; the compiler runs it, reads the page again and reports what actually changed. Compilation uses no model inference, no site-specific code and no prior visit. To our knowledge it is the first such compiler in operation.

We tested it against the browser tools and agents in common use, as shipped, across three model sizes and three task lengths, with its memory and learning switched off. On long tasks that every system attempted, its default mode completed 76 percent (80 of 105); the best alternative in each comparison completed 46 percent. It was the most stable system tested: it never fell below 55 percent in any condition, while every alternative fell to 28 percent or lower. The cheapest alternative on the same model cost 1.9 to 26 times as much per completed task. On a small model it cost less than every alternative on a frontier model, in 30 of 30 pairings, and matched or exceeded its completion rate in 24. When it reported a long task finished, the judge disagreed less than half as often as with the alternatives (22 against 56 percent). Every result is a first visit, the least favourable setting for the design. We publish the tasks, the judging rules and the outcome of all 3,443 counted runs.

1. Introduction

An agent that operates a website has to answer two questions at every step: what can be done on this page, and did the last action work? Current systems answer both by having a language model read the page.

The second question is the one that goes wrong quietly. We asked each tool to make an illegal chess move on Chess.com: the site refused it, and the board stayed exactly where it was. Browser Use returned `ok: true` in tools mode, and its agent reported "Moved the white pawn from e2 to e5." Playwright MCP and Browser Harness gave the same reply they give when a move goes through. Protocolis returned `moved: false`. The replies describe different events: the browser received an input, and the game rejected the move. An agent deciding whose turn comes next needs the second fact.

The distinction matters most where a wrong "done" costs money. Take a bank transfer. The browser can report that the Submit button was clicked while the bank shows an error, asks for a second confirmation, or has quietly timed out. An agent that takes the click for the payment tells its user that the money has moved when it has not. An agent that is unsure and clicks again may send it twice. Knowing that the job is finished requires evidence from the page itself: the confirmation, the reference number, the new balance. A model can inspect screenshots or read page snapshots after each step to gather that evidence. Each new decision then incurs the cost of interpreting the page again, even when the next step is as routine as picking an account from a list.

Protocolis moves that work into a web compiler. The compiler derives operations from the page as it stands. A model can ask for a filter, a form field or a piece of text through those operations. The compiler observes the result and tells the model what the site actually did. We use *solid-state web execution* to mean this division of labour: programmed rules interpret the live page, while a model chooses what to do with the resulting operations and evidence. The site can change after every action.

To our knowledge, Protocolis is the first operationalised first-contact web compiler for AI agents: the first system in real use that compiles a web application into verified operations when the agent arrives, with no model inference in the compilation and no site-specific setup. How it does this is the subject of two pending patent applications.

This paper asks whether that division pays for itself **on the first visit**. Prior exploration, site-specific adapters and remembered workflows can make a familiar site cheaper to operate. Here, the agent must discover what it can do during the task itself. That makes it the least favourable setting for the design: nothing is reused. We test the compiler both as a tool for an existing agent and as part of a complete agent. The shared-agent experiment is especially informative because it holds the outer decision loop fixed while changing the browser interface.

Each system we compared has conditions where it does well and conditions where it falls away. Across the task lengths and model sizes each ran on the same model as Protocolis, Playwright MCP completed between 22 and 100 percent of tasks, Browser Use (tools mode) between 28 and 100, Browser Harness between 11 and 95, Browser Use (agent) between 20 and 100, and Stagehand between 22 and 100. Protocolis never fell below 55 percent in any condition. In that sense it works as a stabilisation layer between an agent and the web: the result depends less on the agent's model and on the length of the task.

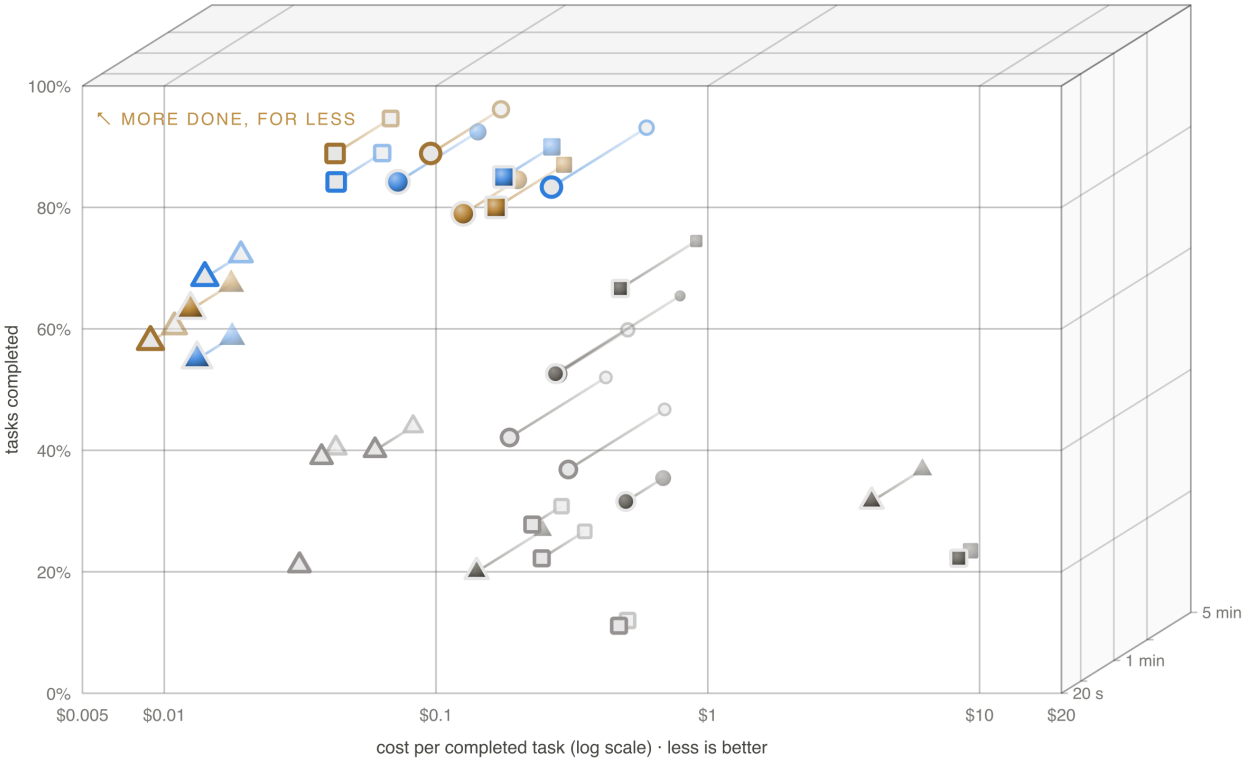


Figure 1: long tasks. Tasks completed against cost per completed task (log scale), with time as depth. Gold is Protocolis Adaptive, blue is Protocolis Standard, and grey is every other system. A triangle, circle or square is the small, strong or frontier model. A ring is a tool driven by the outside agent; a solid mark is an agent. Each system is drawn on the front plane at its completion and cost, and joined by a line to its position in depth, which is its time per completed task: further back is slower. Every mark is a row of Table 3 in Appendix A.

Our contributions are:

1. A head-to-head evaluation against browser tools and browser agents in common use, with every product run as shipped, across three task lengths and three model tiers.
2. Every result measured on first contact: in no run did the system have memory of the site or any site-specific code. Section 5 sets out what volume adds to it: reuse on repeat visits, rules that apply across sites, learning from outcomes, and a per-task cost gap that multiplies with the number of tasks.
3. A count of wrong finishes for every system, and an analysis of our own failures by class.
4. A public replication kit with the task texts, the outcome of every counted run for every system and the judging rules, and access to the system for black-box replication.

2. The system: what the agent can ask the browser to do

We describe the system by what an evaluator can observe when using it. How it produces this behaviour is not disclosed here. It works on a site it has never visited: there is no recording, no site-specific code and no training run, and no model call is needed to compile a page or to offer the agent its operations.

On reaching a page, the agent receives named operations with typed arguments. It can ask for text, choose a control, apply a group of filters or fill a section of a form in one call. When several fields change together, the reply reports the outcome for each. The model sees control names and the text it requests. The compiler handles the raw page and its subsequent changes.

A filter illustrates the contract. A site may offer the exact value requested, approximate it, or omit the filter altogether. Those outcomes call for different decisions. The compiler looks at the resulting page and reports which outcome it can support. If several controls fit a request, the reply identifies the ambiguity. If only part of a list can be read, the result identifies what is missing. In this paper, *verified* describes that check against observable page state.

The same distinction applies to writes. When the effect of a submission is uncertain, the system reports the uncertainty and withholds an automatic retry. If a page crashes or is replaced, the system reports recovery to the agent. At the end of a task it also checks the proposed answer against the page and can return a contradiction or an incomplete list for another attempt. The model never holds secrets: the agent can sign up, sign in and use passwords, payment cards and API keys that its model never receives, and a page that tells the agent to put a stored secret somewhere else does not get it. A second factor or a consent step goes to the user. These are features of the interface. The benchmarks below measure task outcomes.

Protocolis exposes the same operations in two modes. In **Standard**, the agent's model chooses every operation. In **Adaptive**, the default, a smaller model handles routine choices and calls on the agent's model when it is unsure. Neither mode uses model inference to compile the page. A lower-level fallback is available where the compiler lacks a rule.

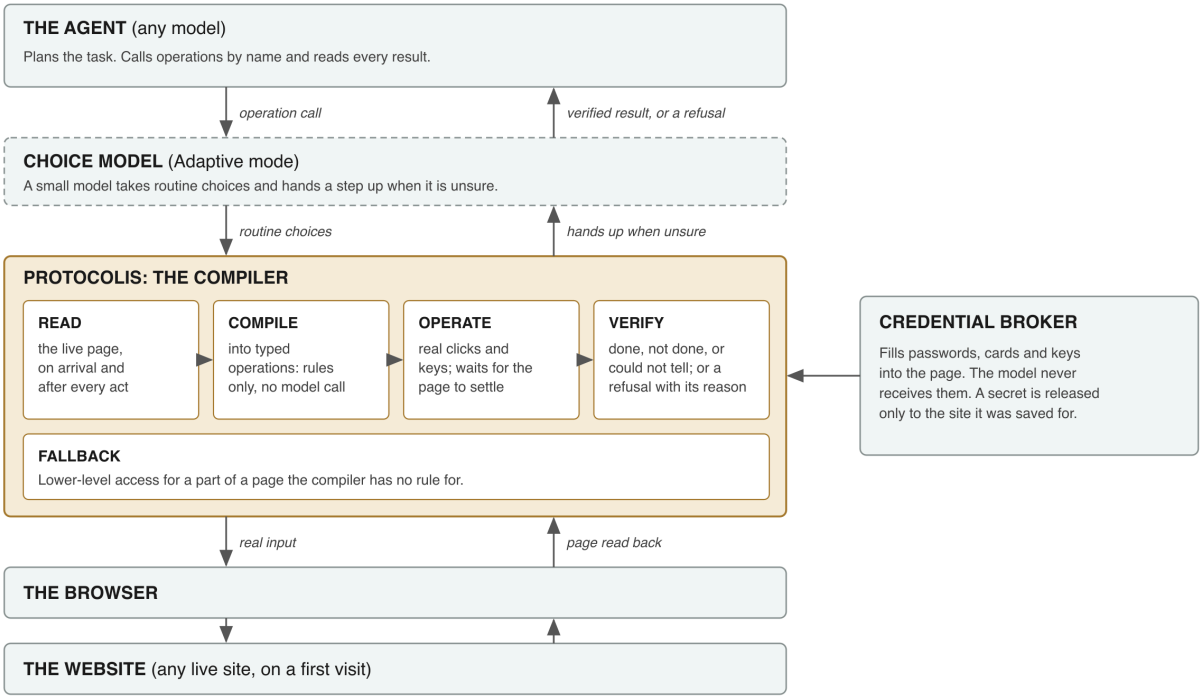


Figure 2: the system as an evaluator sees it. Standard sends every choice of operation to the agent's model; Adaptive can settle routine choices with a smaller model.

3. Results

Every figure in this section is generated from the published record (Section 6.4). A cell is one category, Tools or Agents, at one task length and one model size, so there are 18 cells; a row is one system in a cell. Appendix A lists every cell.

Protocolis is the most stable system tested. Over every combination of task length and model size in which a system ran the same model as Protocolis, Protocolis never completed fewer than 55 percent of the tasks (its four configurations bottom out at 55 to 68 percent); every other system's lowest was 11 to 28 percent. The distance between its best and its worst condition was 32 to 45 points; for every other system it was 72 to 84. At each other system's own best condition, Protocolis completed as large a share of the tasks, or a larger one.

3.1 Completion

We tested two ways to use Protocolis. As a tool, it shared one outside agent and the same limits with Playwright MCP, Browser Use and Browser Harness. As a complete agent, it ran its own loop beside Browser Use and Stagehand running theirs. Each category ran short, medium and long tasks at three model sizes. The widest difference appeared where several conditions had to hold at the end of a task. Table 1 compares systems on identical long tasks: within each category and model size, only the tasks that every system attempted. It keeps Protocolis in its default mode throughout and, for the alternatives, takes the highest-scoring system separately in each comparison.

Table 1: long tasks completed on the tasks every system in a comparison attempted. The total counts attempts across the six comparisons.

Category	Model size	Shared tasks	Protocolis Adaptive, completed	Best alternative, completed	Best alternative
Tools	Small	16	10	7	Playwright MCP and Browser Use (tools mode) (tie)
Tools	Strong	18	16	9	Playwright MCP
Tools	Frontier	16	14	5	Browser Use (tools mode)
Agents	Small	18	11	5	Stagehand
Agents	Strong	19	15	10	Browser Use (agent)
Agents	Frontier	18	14	12	Browser Use (agent)
Total		**105**	**80 (76%)**	**48 (46%)**	

The table holds Protocolis to its default mode. Taking the better of its two modes in each comparison raises its total on the shared tasks to 83 of 105, and it leads in every one of the six comparisons. On each system's full set of long tasks, which differs slightly from system to system, the better Protocolis mode completed 90 of 113 (80 percent) and Adaptive alone 86 of 113 (76 percent), against 51 of 113 (45 percent) for the best other system in each.

One fixed-mode comparison makes the tradeoffs concrete. In the long Agents comparison with Claude Haiku 5.5, Protocolis Adaptive completed 12 of 19, Stagehand 6 of 19, and Browser Use 4 of 20. Model spend per completion was $0.0125, $4.0056, and $0.1411, respectively. Mean times among completions were 46.7, 58.0, and 79.7 seconds. One task asked for a Coursera course on AI ethics lasting under five weeks with a rating of at least 4.5 stars, along with the course and instructor names. The judge accepted Adaptive’s run on that task. Neither competing agent reported it complete. The individual task illustrates the kind of evidence a long result required, while the comparison uses all counted runs in those rows.

In the Tools category, where one outside agent drives every tool, there are 9 cells. Taking the better of the two Protocolis modes in each, it completed a larger share of the tasks than every other tool in 4 and the same share as the best of them in 4; it was behind in 1: Tools, medium, small (77 of 80 against 78 of 80 for Browser Use (tools mode)).

In the Agents category there are 9 cells (three task lengths by three model sizes). Taking the better of the two Protocolis modes in each, it completed a larger share of the tasks than every other agent in 6 and the same share as the best of them in 3; in none was it behind. On short and medium tasks the best Protocolis mode and the best other agent are within 5 percentage points of each other in every cell. Table 1 gives the long-task counts, and Appendix A shows both modes in every cell.

3.2 Stability across model sizes

Between the strong and the frontier model on long tasks, Protocolis's completion moved by at most 1.1 percentage points in each of its 4 rows. The other systems moved by 9 to 31 points, and 4 of the 5 (Playwright MCP, Browser Use (tools mode), Browser Harness, Stagehand) completed fewer long tasks on the frontier model than on the strong one.

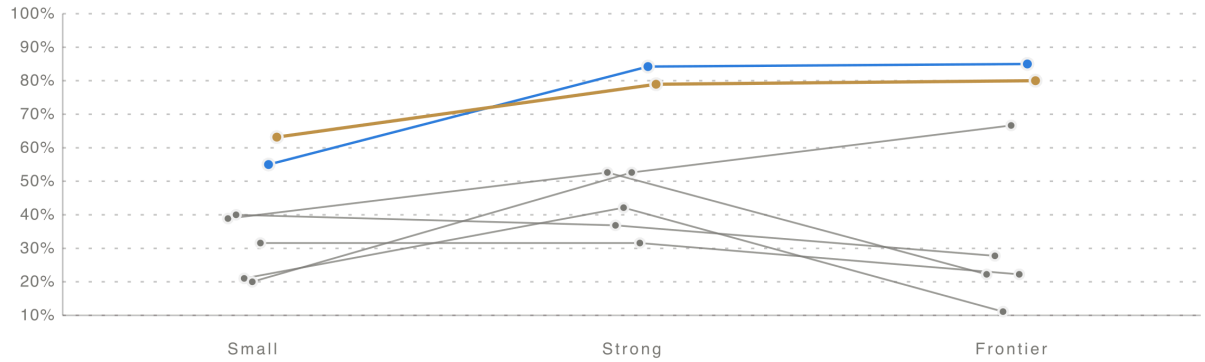


Figure 3: long tasks, completion by model size. Protocolis with its own agent (gold Adaptive, blue Standard) and every other system (grey).

Across model sizes alone, Protocolis's completion moved least in 4 of the 6 category and task-length groups. On long tasks in the Tools category, Playwright MCP (22 to 53 percent) and Browser Use (tools mode) (28 to 40 percent) moved less than a Protocolis mode, at a level below Protocolis's lowest (58 percent). On long tasks in the Agents category, Stagehand (22 to 32 percent) moved less than a Protocolis mode, at a level below Protocolis's lowest (55 percent).

A small model on Protocolis against a frontier model elsewhere. Pairing each Protocolis row on the small model with each other system in its category on the frontier model gives 30 pairings across the three task lengths. Protocolis on the small model cost less per completed task in 30 of 30, matched or exceeded its completion rate in 24, and took less time per completed task in 22. All three held together in 18. The pairings in which its completion rate was lower: Tools, medium tasks, against Playwright MCP (2 pairings); Tools, medium tasks, against Browser Use (tools mode) (2 pairings); Agents, long tasks, against Browser Use (agent) (2 pairings).

On long tasks in the Tools category, Protocolis on the small model (Claude Haiku 5.5) completed 58 percent (11 of 19) in Adaptive and 68 percent (13 of 19) in Standard. No other tool exceeded 28 percent (5 of 18, Browser Use (tools mode)) on the frontier model (GPT-6.1 Sol), and the cheapest of them cost 16 to 25 times more per completed task. On long tasks in the Agents category, Protocolis on the small model (Claude Haiku 5.5) completed 63 percent (12 of 19) in Adaptive and 55 percent (11 of 20) in Standard. The best other agent completed 67 percent (12 of 18, Browser Use (agent)) on the frontier model (GPT-6.1 Sol), and the cheapest of them cost 36 to 38 times more per completed task.

3.3 Cost per completed task

In every one of the 18 cells where Protocolis and another system ran the same model, the lower-cost Protocolis mode cost less per completed task than the lowest-cost other system. The ratio runs from 1.9 times (Tools, medium, small, against Playwright MCP) to 26 times (Agents, short, strong, against Browser Use (agent)). By task length: short 2.8 to 26 times, medium 1.9 to 13 times, long 2.0 to 11 times. Those ratios are against the lowest-cost other system in each cell; against the rest the gap is wider. The widest is in Agents, long, small: Stagehand cost $4.01 per completed task, completing 6 of 19, against $0.0125 for Protocolis Adaptive, completing 12 of 19: 320 times as much. Section 5 gives the gap per million tasks. Appendix B gives tokens per completed task for every cell.

In Adaptive with its own agent, 201 of 359 runs on the short and medium rows (56 percent) finished without a call to the agent's model. Of the 201 finishes the small choice model made itself, none was judged wrong. Run with its own agent, the agent's model was called 0.7 times per task in Adaptive against 3.1 in Standard.

3.4 Time per completed task

Protocolis was the fastest system in 11 of the 18 cells. In 5 of the others, all on long tasks, each faster system completed 2 to 10 tasks, against 11 to 17 for Protocolis in its faster mode. In Tools, long, strong the difference is 0.5 seconds. In the remaining 2 (Tools, short, strong; Tools, medium, strong), Playwright MCP was 3 to 4 seconds faster at equal completion. On short and medium tasks the faster Protocolis mode took less time per completed task than the fastest other system in 10 of 12 cells, by up to 4.5 times. On long tasks it was faster in 1 of 6 cells. Table 6 lists every cell in which another system was faster, with both completion counts. Time is the mean duration of completed runs (Section 6.4), so those cells are read beside the completion counts.

3.5 Wrong finishes

Table 2: runs in which a system reported completion and the judge disagreed.

System	Category	Short and medium: judged not complete, of reported complete	Long: judged not complete, of reported complete
Protocolis Adaptive	Tools	6 of 299 (2.0%)	11 of 54 (20.4%)
Protocolis Standard	Tools	6 of 300 (2.0%)	11 of 55 (20.0%)
Protocolis Adaptive	Agents	1 of 359 (0.3%)	14 of 57 (24.6%)
Protocolis Standard	Agents	2 of 379 (0.5%)	13 of 57 (22.8%)
Playwright MCP	Tools	13 of 357 (3.6%)	34 of 54 (63.0%)
Browser Use (tools mode)	Tools	8 of 298 (2.7%)	26 of 46 (56.5%)
Browser Harness	Tools	19 of 293 (6.5%)	25 of 39 (64.1%)
Browser Use (agent)	Agents	18 of 320 (5.6%)	21 of 47 (44.7%)
Stagehand	Agents	4 of 309 (1.3%)	16 of 32 (50.0%)

A run counts as reported complete when the system's own record says it completed and its answer does not say it was blocked. On long tasks Protocolis reported a finish that the judge did not accept in 49 of 223 (22.0 percent) of its reported finishes; the other systems in 122 of 218 (56.0 percent), from 45 percent (Browser Use (agent)) to 64 percent (Browser Harness). On short and medium tasks the figures are 15 of 1,337 (1.1 percent) for Protocolis and 62 of 1,577 (3.9 percent) for the other systems. The largest single count: Playwright MCP on the frontier model reported 18 long tasks complete, and the judge did not accept 14 of them. Stagehand's long count includes 1 run that the judge accepted and that went over the cost limit (Section 6.3).

3.6 Where another system leads

Across the 18 cells, taking the better Protocolis mode on each metric, another system leads on completion in 1, on cost in 0 and on time in 7. Table 6 lists each one. Each time row gives both completion counts. Appendix A shows both modes in every cell.

4. Failure analysis

The long set holds 54 of the 70 Protocolis runs in Appendix A that were not judged complete. Each of the 54 was assigned to one class from its own record.

Class 1: finished and rejected, or out of limit. The system worked the page through to an answer that the judge did not accept, or reached its time or turn limit. This class holds 25 of 54.

Class 2: coverage. At some step the system reported that it was unsure, refused the step, or found itself repeating. This is how a page pattern outside the compiler's current rules appears in a run's record. This class holds 23 of 54.

Class 3: outside the system. The site refused the session, by a challenge or an access denial, or a network or model-provider error ended the run. This class holds 6 of 54.

Coverage failures are addressed by adding rules, not by per-site code.

5. What scale adds

The results above describe a first visit, at the volume of a benchmark. Several properties of the design grow with volume. None of them is measured in this paper, and memory and learning were off in every run. We state each as a property to be tested, with the arithmetic that follows from the measured figures.

Warm knowledge. Every run here began with no memory of the site. The system can keep what it compiled for a site and reuse it on the next visit. A first visit is therefore the worst case, and it is paid once per site. The more traffic a deployment carries, the larger the share of its visits that are repeat visits.

Reliability that compounds, by two routes. The first route is the compiler's rules. A coverage failure (Section 4, class 2) points to a page pattern, not to a site. The remedy is a rule for that pattern, and the rule then applies on every site that uses it. Web applications are assembled from a limited stock of components and conventions, so one rule retires a class of failures. Per-site adapters work the other way: their upkeep grows with the number of sites.

The second route needs no developer. The system learns from outcomes: it records which of its checks held when an act took effect, keeps the checks that go on holding, and drops the ones the page's evidence stops supporting. What it learns on one site can carry to others. More traffic feeds both routes: it brings the remaining patterns to light sooner, and it gives the learning more outcomes to learn from.

Cost that follows operations, not pages. The cost gap in Section 3.3 is per completed task, so it multiplies with volume. At the measured cost per completed task with the frontier model, one million completed tasks cost: on medium tasks, $5,200 with Protocolis against $68,500 with the lowest-cost other agent (Browser Use (agent)); on long tasks, $166,300 with Protocolis against $477,300 with the lowest-cost other agent (Browser Use (agent)). This is arithmetic on the measured figures, not a measurement at that volume.

A deployment at scale also chooses its model. Section 3.2 shows what the choice costs here: on the small model, Protocolis cost less than every other system on the frontier model in every pairing, and matched or exceeded its completion rate in most of them.

6. Evaluation and replication

We evaluated task completion, model cost and time, and asked how these measures changed with task length and model tier. We also measured first-contact performance and wrong finishes, where a system reported a finish that the judge found incorrect. A report that the system was blocked was not counted as a wrong finish.

6.1 Comparison groups and systems

We separated the evaluation into two categories. In the Tools category, one outside agent loop drove each browser tool with the same model, turn limit and time limit. The tools were Playwright MCP 0.0.84, Browser Use 0.13.13 in tools mode, Browser Harness 0.1.13, and Protocolis Adaptive and Standard. In the Agents category, each

product used its own agent loop and limits. The systems were Browser Use 0.13.13, Stagehand 3.7.3, and Protocolis Adaptive and Standard. The analysis ranks systems within each category.

Each competing product ran at its named release with documented defaults. We did not tune competing systems. Each product kept its own reasoning setting, and the outside agent used the provider's default. The published record identifies the model, reasoning setting sent to the provider and driving agent for each row. Protocolis had no configuration for any site in the task sets.

6.2 Tasks and operating conditions

The short set contained 20 single-objective tasks on stable public pages with exact gold answers. The medium set contained 40 multi-step tasks on live sites. Both sets were written for this evaluation. Set sizes count distinct tasks; a row's count in the tables is its attempts, and some rows ran every task two or three times. The long set targeted 20 tasks from WebVoyager [He et al., 2024] and Mind2Web-Live [Pan et al., 2024], each with several conditions that had to hold in the final state.

All tasks ran on the live public web. No system used replay or state learned from an earlier run of the same task. Every Protocolis run was a first visit. We disabled site memory and learning from outcomes. Protocolis had no adapter, recording or code for any task site. The short pages were likely to have appeared in model training, so the medium and long sets provide the first-contact test without relying on the short set for generality.

Two Amazon tasks were excluded for all systems because the site refused every system. Where a replacement was run, it came from a list fixed before any run. An individual long run was excluded when its row's published record identified a site rate limit or robot check, or a test-side fault that lost the record. Other site refusals remained failures. Long cells therefore contained 18 to 20 runs and did not always contain identical tasks. The strong tier had 19 tasks, and 18 long tasks were common to all three tiers. For the matched long-task comparison we restrict each category and model tier to the task IDs that every system in that comparison attempted.

Workers used the same network exit and ad blocking, with one task at a time per worker. Two worker classes were used and the record identifies the class for each row. Class A measured about 1.2 times faster as a machine, so differences of that size can be explained by worker class when rows ran on different classes.

6.3 Model tiers and judging

The small tier used GLM-5.3-flash for short and medium tasks and Claude Haiku 5.5 for long tasks. The strong tier used Grok 4.6 throughout. The frontier tier used GPT-6.1 Sol, with Claude Opus 4.6 also used for some short-task systems. Where a cell includes both frontier models, the result tables mark the second model and do not rank it against the first.

For short and medium tasks, completion required the system to finish on the required page, the page to show an accepted answer, and the system's own answer to be accepted. Every attempt was counted, except a medium run whose evidence could not be captured; that rule applied to every system, and the row's record names each omitted attempt. The Protocolis Tools rows on short tasks were counted as published using the page check. In two Protocolis Agents rows on medium tasks with the strong model, both runs of one task ended at a robot check and were measured again. The second measurement was counted.

Long tasks used WebJudge [Xue et al., 2025] with the upstream Online-Mind2Web prompts and strict scoring. One judge model (GPT-5.6 Luna, medium reasoning) assessed every system's runs using action histories and screenshots, without seeing the agent's final answer. Key points were written once per task and shared across systems, and each system supplied one frame per operation. A frontier-model long run that spent more than $2.00

on model calls counted as incomplete, with its spend retained in cost. This rule removed one otherwise accepted Stagehand run.

Judging routes and screenshot selections were not identical across all rows. To measure the resulting variability, 306 long runs were judged again by the same model using a different screenshot selection or API route. The verdict changed for 26 runs, or 8.5%. For a cell of 20 tasks, a difference of about two tasks or less should therefore be read as a tie. On eight repeated judgments with identical input, no verdict changed.

6.4 Measures and provenance

Completion is the number judged complete divided by attempts, reported with the count. Cost per completion is total model spend across the set divided by completions, so failures contribute to cost. Calls are counted at billed prices, including auxiliary calls, except for an omission marked by row in the record. In most Protocolis rows, the system's own small-model calls are included in token counts but not priced. Where priced, that model costs $0.042 per million input tokens. A sensitivity calculation charging every Protocolis row's entire token count again at that price adds $0.0002 to $0.0140 per completion. Under that calculation Protocolis remains lowest in every cell, while the smallest reported cost ratio falls from 1.9 to 1.4.

Time per completion is the mean duration of completed runs, with medians in the record. The clock runs from tool readiness to the verified answer on the same host. Where a competing system reports its own clock, that clock is used and identified in the record. Unlike cost, time excludes failures, so a system completing only its quickest tasks can have a short reported time.

Each measurement window was registered before launch with its build, task set and planned comparison. Protocolis measurements ran from 9 to 19 November 2026 on the build current on each run date. Competing systems were measured from 9 to 20 November 2026 at the versions named in Section 6.1. The published record identifies the build or version, measurement window, model, judging rule, counts and cost basis for each row.

6.5 Replication and access

The replication kit contains 81 system-level results and 3,443 counted runs. It includes the task texts, both the self-reported and judged outcome of each run, and the criteria used to judge each long task. Readers can recompute completion figures from the kit and check cost and time against the record. Recorded runs and competing-system configurations are available on request.

Protocolis is available for black-box testing through a hosted evaluation platform. Readers can request a key and test sites of their choosing without a non-disclosure agreement. The source code, rules and internal mechanisms remain proprietary. External evaluators can test the behaviour on new sites.

6.6 What we did for the other systems

We built one of the systems compared. At each choice in the design we therefore tried to give the other systems the benefit.

- Each product ran at a named release, with its documented defaults and its own reasoning setting.
- In the Tools category one outside agent, with the same model and the same limits, drove every tool, ours included.
- A task whose site refused every system was left out for every system.
- Where a competing system reports its own clock, we used it.

- Time is reported for completed runs only. That favours a system that completes only its quickest tasks. We kept the definition and print each completion count beside it.
- On long tasks one judge model ruled on every run and never saw an agent's answer text. Every system supplied the same kind of evidence.
- In the count of wrong finishes, a run in which a system said it was blocked is not counted against it.
- Every cell in which another system leads is listed (Section 3.6).
- The outcome of every counted run is published and the recorded runs are provided on request, so any completion figure can be recomputed.

7. Related work

Web agents differ in how they obtain the actions available to them and what preparation those actions require. Screenshot-based agents choose actions from visual representations of a page. SeeAct [Zheng et al., 2024] studied the grounding of multimodal model actions on live pages. Other systems expose structured views. Playwright MCP returns accessibility snapshots, Stagehand provides model-resolved act, observe and extract primitives, and Notte presents typed actions. Agent-E [Abuelsaad et al., 2024] combines a distilled page view with a planner and browser agent. It reports 73.1 percent completion on WebVoyager, as well as task time, model calls and unnoticed failures. These approaches differ substantially, but each uses a model to interpret the page during the task.

Operations prepared in advance

Another line of work gives agents operations obtained from site interfaces or prior preparation. Song et al. [2024] report 38.9 percent on WebArena using a hybrid of documented APIs and browsing, compared with 14.8 percent using browsing alone. The authors supplied missing APIs for one of the five sites. SkillWeaver [Zheng et al., 2025] explores sites and distils reusable APIs, reporting relative gains of 31.8 percent on WebArena and 39.8 percent on 57 tasks from four live sites. WALT [Prabhu et al., 2025] reverse-engineers site functions into tools. ActionEngine [Zhong et al., 2026] builds a state-machine map offline and refines it in a warm-up run before generating a program for each task. It reports 91.2 percent success on 655 tasks across four WebArena sites, compared with 82.7 percent for Claude Code with Playwright MCP on the same model, with lower reported latency and cost. Its offline crawl took 36 to 260 minutes per application.

Operations can also come from captured requests. Unbrowse [Tham et al., 2026] learns internal routes from browsing traffic and reuses them, with browser fallback. On retrieval tasks across 94 live domains from its index, it reports a 950 ms mean with a fully warmed cache, compared with 3,404 ms for scripted Playwright extraction. On 20 previously unseen domains, the first request had a median time of 8.2 seconds and route extraction succeeded on 18. Answer correctness was unreported. Site2CLI captures traffic for individual operations and promotes an operation after five successes without failure, where success means a call returned without error. API Anything learns a request using two example inputs and checks a write by reading back its result. Its repository reports graded answers, time, cost and tokens on three sites. OpenCLI instead provides adapters for named sites alongside generic browser commands, without a published evaluation.

Other systems reuse earlier experience. Agent Workflow Memory [Wang et al., 2024] induces workflows from trajectories, while ContractSkill [Lu et al., 2026] makes skills explicit and repairable. TraceCompiler [El Yadouni and Li, 2026] compiles earlier API-call traces into mostly deterministic workflows. Agent JIT [Winston et al., 2026] synthesises tools from application traces and compiles each task into code. On 37 tasks across five applications, it reports a 10.4-fold speed-up and 28 percent higher accuracy than Browser Use, with per-

application setup intended for repeated use. WebMCP and CI4A [Qiu et al., 2026] take a cooperative route, exposing tools through the site or its framework. CI4A reports 86.3 percent on 182 WebArena tasks whose front end was rebuilt with its library.

Operations on a first visit

Ghost is the closest documented comparison for first visits. Its documentation describes generating site-specific tools from page structure using heuristics, refining them with a model in the background and caching them. It reports 94.6 percent success over 1,637 proprietary tests, but does not describe their tasks or method, and describes no check of an action's effect. Jiang et al. [2026] argue for typed actions published by developers. Piet et al. [2026] propose compiling a task into a program before reading untrusted page content and identify the lack of typed interfaces with known effects as a barrier. Compiling a task into a program and compiling an application into operations are distinct, and complementary. Mansoor [2026] studies verified structured tool calls under non-atomic failures, a related verification problem outside the web-page setting considered here.

Evaluating across systems

The evaluation builds on established benchmarks. Mind2Web [Deng et al., 2023] introduced more than 2,000 tasks on 137 real sites, WebArena [Zhou et al., 2023] offers self-hosted sites, and WebVoyager [He et al., 2024] evaluates on 15 live sites. Online-Mind2Web [Xue et al., 2025] re-evaluated agents on 300 tasks across 136 live sites and introduced the WebJudge protocol used here for long tasks. BrowserGym [Le Sellier De Chezelles et al., 2024] provides a common interface across benchmarks, while cua-speedrun [Aggarwal et al., 2026] standardises speed and cost measurement for computer-use agents. Tool-maintainer comparisons also report completion, cost and time across browser setups. Together these studies motivate examining first-visit task results, cost and time per completion, and wrong finishes across models and task lengths. They do not make results from different sites, preparations and evaluation protocols directly interchangeable.

Position of this work

To our knowledge, the system evaluated here is the first just-in-time compiler that turns a web application into a verified protocol for AI agents, with no model inference in the compilation and no site-specific setup. Each qualifier matters. Structured snapshots exist without model-free compilation into operations. Typed actions exist with a model in the conversion. Tools generated from the page by heuristics exist, refined by a model and without verified results. Operations drawn from request capture, exploration, earlier runs or site adoption exist with per-site preparation or, for request capture, at a first visit without a verified task result. Stored requests learned without a model exist after a learning step for each operation on example inputs; the one that checks a write does so by reading the result back. Compilation of a task into a program exists and presupposes the operations. Verification of a tool call with three outcomes, true, false or unknown, exists for structured tool calls [Mansoor, 2026]; we found no system that applies it to web pages. What sets the system apart from tools generated from the page is the verified result defined in Section 2.

Read together, the published evaluations leave five things open, and this evaluation is built to address them: results on a first visit, with no preparation for the site; the live web beyond benchmark sites; cost and time per completed task, not per attempt; shipped tools and shipped agents measured together under one protocol across task lengths and models; and the rate of wrong finishes compared across products. Table 7 in Appendix C sets the lines of work side by side.

8. Limitations

- Independent replication has not been done yet. The records and the access described in Section 6.5 are there for it.
- A long cell holds 18 to 20 runs, so a single cell is a small sample. The pooled long-task result is the one to rely on: 90 of 113 against 51 of 113 for the best other system in each cell, a lead of 35 percentage points.
- Live sites change. A task valid on the run date may not be valid later.
- All runs left through one network exit. Some sites treat traffic differently by origin.
- The system is closed. Behaviour can be tested and mechanism cannot.
- Repeat visits are not evaluated (Section 5). Site memory was off in every run, so these results describe a first visit only.
- Tasks are in English on desktop layouts.
- The other systems ran at their defaults and might do better with expert configuration. Protocolis ran with no configuration for any site.
- The short set uses pages likely to be familiar to models, for every system alike.
- We did not run several systems from Table 7: Unbrowse, ActionEngine, Agent JIT and the skill-learning agents. They are research artefacts or need a preparation pass per site, and we compared shipped products as shipped. Unbrowse reports faster times on retrieval tasks served from a warmed cache, which is a different setting from the first visit measured here.

9. Use in practice

Outside the evaluation, the system is in daily use. Examples include filing a patent application through the patent office's online system, obtaining a tax identification number through the tax authority's online application, and routine daily work in freight dispatch offices that pay to use it. These are observations, not measurements, and we draw no quantitative claim from them.

10. Conclusion

A browser agent needs to know both what it can do on a page and what happened when it tried. We compared a system that compiles web pages into verified operations, without a model reading the page, against browser tools and agents in common use. To our knowledge it is the first operationalised first-contact web compiler for AI agents. Across 18 comparisons on the same model, Protocolis had the lowest cost per completed task in every one and matched or exceeded the best other system's completion rate in 17 of 18, with its widest completion margins on long, multi-step tasks. The first-contact results on long tasks and the list of cells where the system is behind are, in our view, the parts of this evaluation most worth replicating. Across every task length and model size it kept the highest floor and the narrowest range of any system tested, and on the same model it matched each alternative on completion where that alternative was strongest.

Every figure here is a first visit with memory and learning off. A deployment at volume starts from these figures and adds to them: repeat visits reuse what was compiled, each general rule applies on every site, the system learns from outcomes, and the per-task cost gap multiplies by the number of tasks (Section 5). Those effects are the next thing to measure.

Protocolis is available for evaluation at protocolis.co.

Use of AI tools

Generative AI tools were used in drafting and editing the text, in checking the related work against its sources, and in writing the code that generates the tables and figures from the published record. I reviewed the paper and take full responsibility for its content.

References

- Abuelsaad et al. Agent-E: From Autonomous Web Navigation to Foundational Design Principles in Agentic Systems. arXiv:2407.13032, 2024.
- Aggarwal et al. cua-speedrun: Standardized Benchmarking of the Speed of Computer-Use Agents. arXiv:2609.40284, 2026.
- Deng et al. Mind2Web: Towards a Generalist Agent for the Web. arXiv:2306.06070, 2023.
- El Yadouni, Li. TraceCompiler: Skill-Guided Mining and Compilation of LLM Agent Traces into Mostly Deterministic Workflows. arXiv:2608.02680, 2026.
- He et al. WebVoyager: Building an End-to-End Web Agent with Large Multimodal Models. arXiv:2401.13919, 2024.
- Jiang et al. Web Agents Should Use Typed Actions Instead of Click-Based Browsing. arXiv:2602.17245, 2026.
- Le Sellier De Chezelles et al. The BrowserGym Ecosystem for Web Agent Research. arXiv:2412.05467, 2024.
- Lu et al. ContractSkill: Repairable Contract-Based Skills for Multimodal Web Agents. arXiv:2603.20340, 2026.
- Mansoor. Verified Tool Calls Improve LLM Agent Reliability Under Non-Atomic Failures. arXiv:2608.02645, 2026.
- Pan et al. WebCanvas: Benchmarking Web Agents in Online Environments (the Mind2Web-Live set). arXiv:2406.12373, 2024.
- Piet et al. Web Agents Should Adopt the Compile-and-Execute Paradigm. arXiv:2605.14290, 2026.
- Prabhu et al. WALT: Web Agents that Learn Tools. arXiv:2510.01524, 2025.
- Qiu et al. CI4A: Semantic Component Interfaces for Agents Empowering Web Automation. arXiv:2601.14790, 2026.
- Song et al. Beyond Browsing: API-Based Web Agents. arXiv:2410.16464, 2024.
- Tham, Mac Gregor Garcia, Hahn. Internal APIs Are All You Need: Shadow APIs, Shared Discovery, and the Case Against Browser-First Agent Architectures. arXiv:2604.00694, 2026.
- Wang et al. Agent Workflow Memory. arXiv:2409.07429, 2024.
- Winston et al. Agent JIT Compilation for Latency-Optimizing Web Agent Planning and Scheduling. arXiv:2605.21470, 2026.
- Xue et al. An Illusion of Progress? Assessing the Current State of Web Agents. arXiv:2504.01382, 2025.
- Zheng et al. GPT-4V(ision) is a Generalist Web Agent, if Grounded. arXiv:2401.01614, 2024.
- Zheng et al. SkillWeaver: Web Agents can Self-Improve by Discovering and Honing Skills. arXiv:2504.07079, 2025.

- Zhong et al. ActionEngine: From Reactive to Programmatic Web Agents via State Machine Memory. arXiv:2602.20502 (v2, 28 Sep 2026), 2026.
- Zhou et al. WebArena: A Realistic Web Environment for Building Autonomous Agents. arXiv:2307.13854, 2023.
- Tool-maintainer comparisons, as fetched 14 Nov 2026: WebdriverIO, Browser Agent Benchmark, https://benchmark.webdriver.io/ (runs of 4 to 9 Oct 2026). LangChain, eight browsing harnesses compared on WebVoyager-100, https://github.com/langchain-ai/deepagents/pull/6868 (opened 9 Oct 2026, closed unmerged). Browser Use, Best browser tool for coding agents?, https://browser-use.com/benchmarks (measured 14 Oct 2026).
- Product documentation and repositories, read 20 November 2026: Playwright MCP, github.com/microsoft/playwright-mcp. Stagehand, github.com/browserbase/stagehand and docs.stagehand.dev. Notte, github.com/nottelabs/notte and docs.notte.cc. Browser Use, github.com/browser-use/browser-use. Browser Harness, github.com/browser-use/browser-harness. Site2CLI, github.com/lonexreb/site2cli. API Anything, github.com/goodnight000/api-anything. OpenCLI, github.com/jackwener/OpenCLI. WebMCP, github.com/webmachinelearning/webmcp. Ghost, gotghost.io.

Submitted by Protocolis

Appendix A. Results for every system

Table 3: every cell. Share of tasks completed (with the count), cost per completed task and time per completed task, by category, task length and model size. A dagger (†) marks a cell run on the model named beside it, not on the model of its column; it is shown and not ranked.

A.1 Short tasks

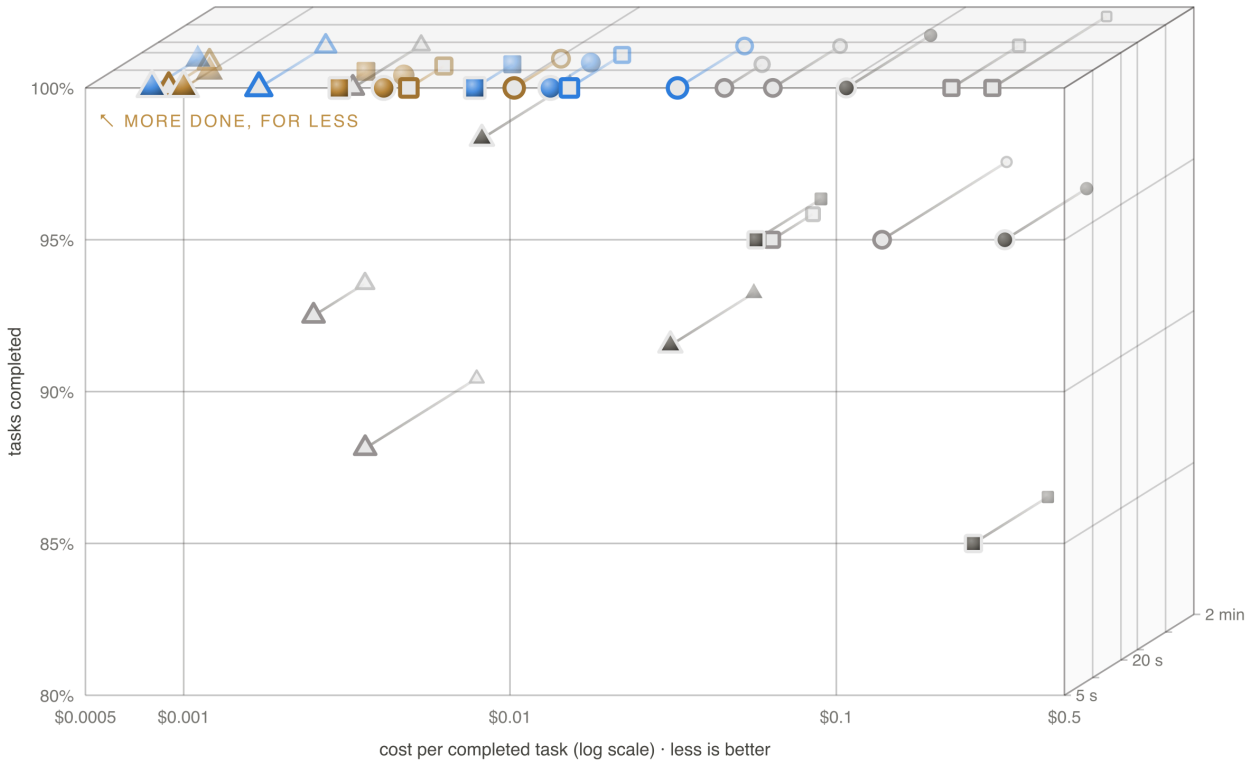


Figure 4: short tasks, drawn as in Figure 1. The completion axis starts at 80 percent.

Tools

System	Small (GLM-5.3-flash)	Strong (Grok 4.6)	Frontier (GPT-6.1 Sol)
Protocolis Adaptive	100% (40/40) · $0.0009 · 14 s	100% (40/40) · $0.0103 · 16 s	100% (20/20) · $0.0049 · 12 s
Protocolis Standard	100% (40/40) · $0.0017 · 26 s	100% (40/40) · $0.0326 · 26 s	100% (20/20) · $0.0152 · 18 s
Playwright MCP	93% (37/40) · $0.0025 · 18 s	100% (40/40) · $0.0454 · 13 s	95% (38/40) · $0.0635 · 14 s
Browser Use (tools mode)	100% (59/59) · $0.0033 · 27 s	100% (20/20) · $0.0639 · 26 s	100% (20/20) · $0.2253 · 26 s †Claude Opus 4.6
Browser Harness	88% (52/59) · $0.0036 · 77 s	95% (19/20) · $0.1382 · 106 s	100% (20/20) · $0.3005 · 83 s †Claude Opus 4.6

Agents

System	Small (GLM-5.3-flash)	Strong (Grok 4.6)	Frontier (GPT-6.1 Sol)
Protocolis Adaptive	100% (60/60) · $0.0010 · 9 s	100% (40/40) · $0.0041 · 8 s	100% (60/60) · $0.0030 · 10 s
Protocolis Standard	100% (60/60) · $0.0008 · 15 s	100% (40/40) · $0.0133 · 14 s	100% (40/40) · $0.0078 · 13 s
Browser Use (agent)	98% (59/60) · $0.0082 · 43 s	100% (40/40) · $0.1074 · 40 s	95% (19/20) · $0.0567 · 25 s
Stagehand	92% (54/59) · $0.0310 · 39 s	95% (38/40) · $0.3285 · 37 s	85% (17/20) · $0.2632 · 31 s

A.2 Medium tasks

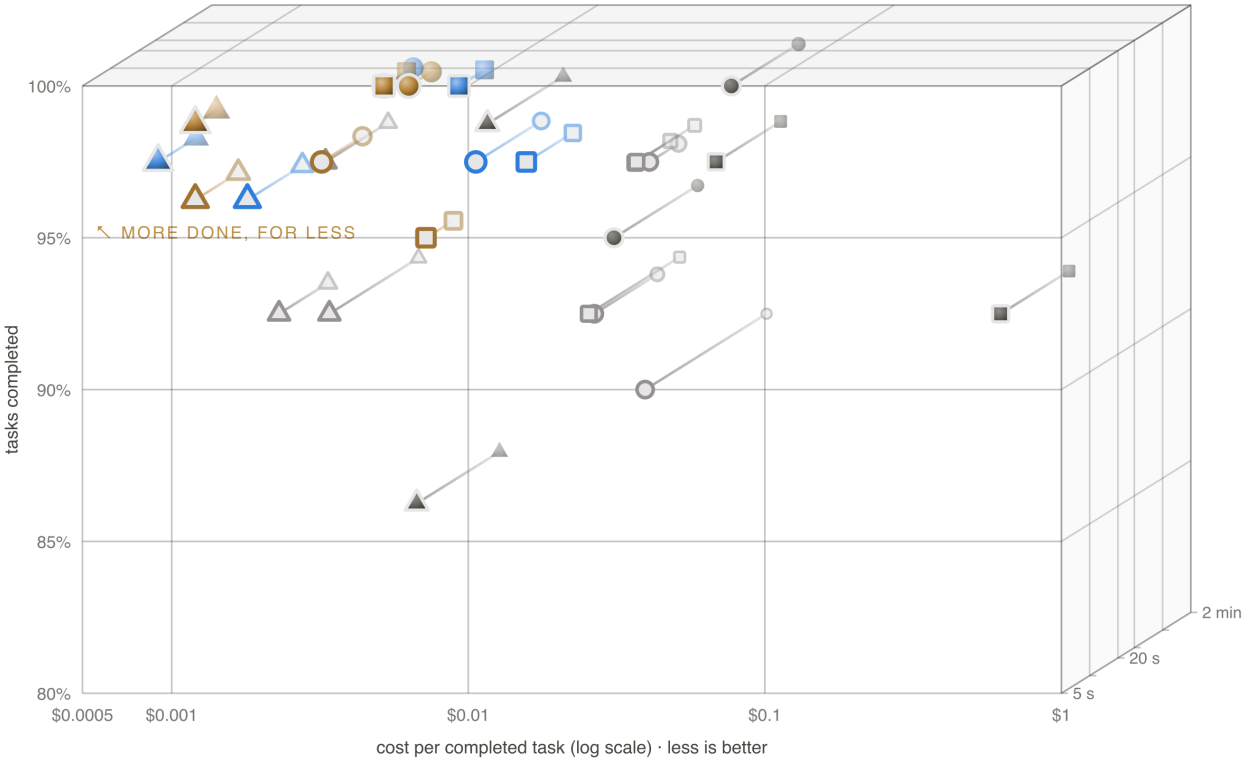


Figure 5: medium tasks, drawn as in Figure 1. The completion axis starts at 80 percent.

Tools

System	Small (GLM-5.3-flash)	Strong (Grok 4.6)	Frontier (GPT-6.1 Sol)
Protocolis Adaptive	96% (77/80) · $0.0012 · 14 s	98% (78/80) · $0.0032 · 14 s	95% (38/40) · $0.0072 · 10 s
Protocolis Standard	96% (77/80) · $0.0018 · 19 s	98% (78/80) · $0.0106 · 25 s	98% (39/40) · $0.0157 · 16 s
Playwright MCP	93% (74/80) · $0.0023 · 17 s	98% (78/80) · $0.0408 · 10 s	97% (77/79) · $0.0366 · 12 s
Browser Use (tools mode)	98% (78/80) · $0.0033 · 23 s	93% (74/80) · $0.0266 · 24 s	98% (39/40) · $0.0369 · 21 s
Browser Harness	93% (74/80) · $0.0034 · 45 s	90% (72/80) · $0.0395 · 98 s	93% (37/40) · $0.0255 · 46 s

Agents

System	Small (GLM-5.3-flash)	Strong (Grok 4.6)	Frontier (GPT-6.1 Sol)
Protocolis Adaptive	99% (79/80) · $0.0012 · 8 s	100% (79/79) · $0.0063 · 9 s	100% (40/40) · $0.0052 · 9 s
Protocolis Standard	98% (78/80) · $0.0009 · 13 s	100% (79/79) · $0.0052 · 10 s	100% (80/80) · $0.0093 · 9 s
Browser Use (agent)	86% (69/80) · $0.0067 · 38 s	95% (76/80) · $0.0310 · 39 s	98% (39/40) · $0.0685 · 24 s
Stagehand	99% (79/80) · $0.0116 · 32 s	100% (80/80) · $0.0771 · 26 s	93% (37/40) · $0.6245 · 27 s

A.3 Long tasks

Figure 1 plots these cells.

Tools

System	Small (Claude Haiku 5.5)	Strong (Grok 4.6)	Frontier (GPT-6.1 Sol)
Protocolis Adaptive	58% (11/19) · $0.0089 · 33 s	89% (16/18) · $0.0955 · 87 s	89% (16/18) · $0.0425 · 64 s
Protocolis Standard	68% (13/19) · $0.0141 · 43 s	83% (15/18) · $0.2660 · 147 s	84% (16/19) · $0.0429 · 52 s
Playwright MCP	39% (7/18) · $0.0379 · 27 s	53% (10/19) · $0.2800 · 87 s	22% (4/18) · $0.2449 · 49 s
Browser Use (tools mode)	40% (8/20) · $0.0596 · 45 s	37% (7/19) · $0.3068 · 150 s	28% (5/18) · $0.2260 · 37 s
Browser Harness	21% (4/19) · $0.0314 · 20 s	42% (8/19) · $0.1865 · 150 s	11% (2/18) · $0.4715 · 24 s

Agents

System	Small (Claude Haiku 5.5)	Strong (Grok 4.6)	Frontier (GPT-6.1 Sol)
Protocolis Adaptive	63% (12/19) · $0.0125 · 47 s	79% (15/19) · $0.1260 · 63 s	80% (16/20) · $0.1663 · 83 s
Protocolis Standard	55% (11/20) · $0.0132 · 42 s	84% (16/19) · $0.0724 · 107 s	85% (17/20) · $0.1776 · 55 s
Browser Use (agent)	20% (4/20) · $0.1411 · 80 s	53% (10/19) · $0.2753 · 270 s	67% (12/18) · $0.4773 · 98 s
Stagehand	32% (6/19) · $4.01 · 58 s	32% (6/19) · $0.4994 · 44 s	22% (4/18) · $8.38 · 26 s

Appendix B. Further comparisons

B.1 Tokens per completed task

Table 4: tokens per completed task: every metered model call in a row's counted runs, divided by its completed runs. A figure marked ≥ is a lower bound.

Cell	Protocolis, tokens per completed task	Fewest among the others	Ratio
Agents, long, small	≥ 103.8k	1,156.9k, Browser Use (agent)	11
Agents, long, strong	≥ 111.8k	≥ 395.8k, Browser Use (agent)	3.5
Agents, long, frontier	≥ 116.3k	295.4k, Browser Use (agent)	2.5
Agents, medium, small	≥ 11.3k	49.8k, Browser Use (agent)	4.4
Agents, medium, strong	≥ 11.9k	56.9k, Browser Use (agent)	4.8
Agents, medium, frontier	≥ 6.2k	46.9k, Browser Use (agent)	7.6
Agents, short, small	≥ 9.3k	56.9k, Browser Use (agent)	6.1
Agents, short, strong	10.6k	58.5k, Browser Use (agent)	5.5
Agents, short, frontier	≥ 7.0k	46.2k, Browser Use (agent)	6.6
Tools, long, small	≥ 75.6k	255.9k, Browser Harness	3.4
Tools, long, strong	302.0k	598.0k, Browser Harness	2.0
Tools, long, frontier	54.6k	398.7k, Browser Use (tools mode)	7.3
Tools, medium, small	≥ 11.3k	39.1k, Playwright MCP	3.5
Tools, medium, strong	≥ 8.7k	48.0k, Playwright MCP	5.5
Tools, medium, frontier	≥ 6.7k	17.0k, Playwright MCP	2.5
Tools, short, small	≥ 7.2k	41.5k, Playwright MCP	5.8
Tools, short, strong	≥ 7.6k	47.4k, Playwright MCP	6.2
Tools, short, frontier	≥ 5.1k	29.6k, Playwright MCP	5.8

In 18 of 18 cells with token counts on both sides, the Protocolis row used fewer tokens per completed task than the other system with the fewest, by 2.0 to 11 times. In 15 of these cells the Protocolis figure is a lower bound, because some calls of the system's own small model were not metered; the record marks each one.

B.2 Completion across model sizes

Table 5: for each system with all three model sizes on a task length, the lowest and highest completion and the gap in percentage points. A dagger (†) marks a range whose frontier cell ran on Claude Opus 4.6.

System	Category	Short	Medium	Long
Protocolis Adaptive	Tools	100% (0)	95 to 98% (3)	58 to 89% (31)
Protocolis Standard	Tools	100% (0)	96 to 98% (2)	68 to 84% (16)
Playwright MCP	Tools	93 to 100% (7)	93 to 98% (5)	22 to 53% (31)
Browser Use (tools mode)	Tools	100% (0) †	93 to 98% (5)	28 to 40% (12)
Browser Harness	Tools	88 to 100% (12) †	90 to 93% (3)	11 to 42% (31)
Protocolis Adaptive	Agents	100% (0)	99 to 100% (1)	63 to 80% (17)
Protocolis Standard	Agents	100% (0)	98 to 100% (2)	55 to 85% (30)
Browser Use (agent)	Agents	95 to 100% (5)	86 to 98% (12)	20 to 67% (47)
Stagehand	Agents	85 to 95% (10)	93 to 100% (7)	22 to 32% (10)

B.3 Cells in which another system leads

Table 6: every cell in which another system leads, taking the better Protocolis mode on each metric against the best other system in its category on the same model.

Cell	Metric	Protocolis	Leader
Tools, short, strong	time	16 s, completing 40 of 40	Playwright MCP 13 s, completing 40 of 40
Tools, medium, small	completion	96% (77/80)	Browser Use (tools mode) 98% (78/80)
Tools, medium, strong	time	14 s, completing 78 of 80	Playwright MCP 10 s, completing 78 of 80
Tools, long, small	time	33 s, completing 11 of 19	Browser Harness 20 s, completing 4 of 19
Tools, long, strong	time	87.3 s, completing 16 of 18	Playwright MCP 86.8 s, completing 10 of 19
Tools, long, frontier	time	52 s, completing 16 of 19	Browser Harness 24 s, completing 2 of 18
Agents, long, strong	time	63 s, completing 15 of 19	Stagehand 44 s, completing 6 of 19
Agents, long, frontier	time	55 s, completing 17 of 20	Stagehand 26 s, completing 4 of 18

Appendix C. Prior work side by side

Table 7: what each line of work needs beforehand, and what its evaluation covers.

Work	Where the operations come from	Needed beforehand	Evaluated on	Reports	Compared with
Agent-E (2024)	none; model reads a distilled page	nothing	WebVoyager	completion, time per task, model calls, unnoticed failures	research agents
Song et al. (2024)	the site's documented APIs	APIs exist	WebArena	completion; steps and cost per task (appendix)	a browsing agent, published WebArena agents
Agent Workflow Memory (2024)	workflows induced from runs	earlier runs on the same site, or training examples	Mind2Web, WebArena	completion, steps	research baselines

Work	Where the operations come from	Needed beforehand	Evaluated on	Reports	Compared with
SkillWeaver (2025)	skills an agent distils into APIs	an agent explores each site	WebArena; 57 tasks on 4 live sites	completion	same agent without skills; published WebArena agents
WALT (2025)	tools reverse-engineered per site	a construction pass per site	WebArena, VisualWebArena	completion, steps	research baselines, Claude computer use
ActionEngine (2026)	programs over a crawled map	an agent crawls each application; warm-up run	655 tasks, four WebArena sites	completion, cost and latency per task, crawl cost	Claude Code with Playwright MCP; AgentOccam
CI4A (2026)	tools built into UI components	the site's framework adopts it	182 WebArena tasks rebuilt on that framework	completion, steps	research agents
Unbrowse (2026)	the site's internal request routes	captured traffic for the route	94 live domains, retrieval tasks	latency (warm and first visit), estimated cost	scripted Playwright extraction
API Anything (2026, open source)	stored site requests, learned without a model	a learning step per operation, on example inputs	three sites, graded answers	time, cost, tokens, answers	a browser agent on Playwright MCP
Site2CLI (2026, open source)	stored site requests	captured traffic for each operation	the project's own tests	a call that returns without error	none
TraceCompiler (2026)	workflows compiled from earlier agent traces of API tool calls	earlier runs of the same intent	AppWorld and T1 (API tools, not web pages)	dependency recovery, call reduction, pass rate on two intents	adjacency and frequency baselines
OpenCLI (open source)	hand-written or agent-authored adapters for named sites, plus generic browser commands	an adapter per site	no published evaluation	none	none
Agent JIT (2026)	tools synthesised from earlier traces; the task is compiled	offline setup per application	37 tasks, five applications	latency, accuracy	Browser Use; provider computer-use agents
Ghost (2026, product)	tools generated from the page by heuristics, refined by a model	nothing	1,637 of its own tests, tasks and method not described	latency, success rate	scripting libraries
Jiang et al.; Piet et al. (2026)	typed site APIs (proposed)	published by the site or built per site	task analysis (Piet); prototype on ten-odd sites and case studies (Jiang)	Piet: none run; Jiang: over 100 tasks completed, no metrics	Jiang: Browser Use in two case studies
Tool-maintainer comparisons (2026)	none; tools that read the page	nothing	100 tasks from one public set	completion, tokens, cost, time	4 to 8 tools under one agent harness
This work	compiled from the page on arrival, no model	nothing	short, medium and long live sets, every run a first visit	completion, cost and time per completed task, wrong finishes	shipped tools and shipped agents