

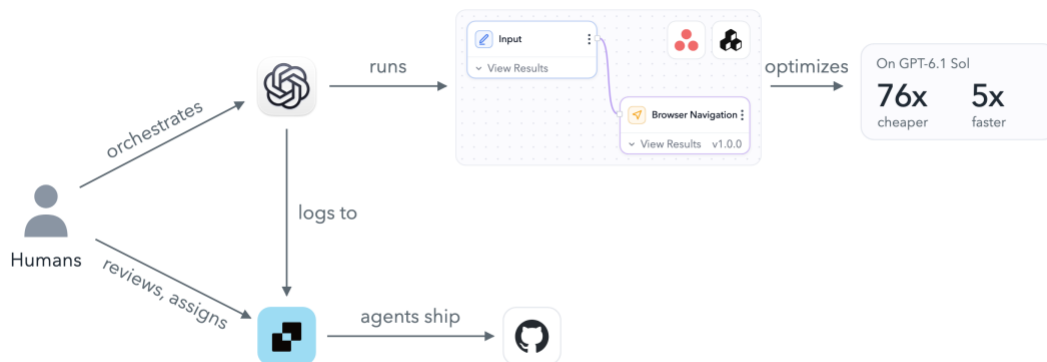
How Asana used Codex to optimize browser agent costs and runtime

A 144-run development study across four models with GPT-6 Astra, from code investigation to measured improvements

Hidalgo, Frank, PhD, 30 September 2026

Executive summary

Asana is building agent automations on StackAI, the agent platform it acquired, and at Asana's scale small inefficiencies add up. On OpenAI's GPT-6.1 Sol, the optimized browser-navigation agent cost 76x less than the original development setup, ran 5x faster and read 89% of its input from cache. The fix was to cache the agent's history without breaking it and to give it a history budget that fits the model, and it held on three other frontier models as well. OpenAI's [GPT-6 Astra](#), working in [Codex](#), investigated the code, implemented the controls, ran the workflows and analyzed the traces. Humans directed the study and reviewed its conclusions. Asana is developing tooling to run similar experiments automatically.



Contents

0	Technical summary	3
0.1	Motivation	3
0.2	Findings	3
0.3	Test design	3
0.4	Results	3
0.5	Scope and limits	4
1	Introduction	5
1.1	Prompt caching in brief	6
1.2	Why the cache misses in agents	6
2	Results and discussion	7
2.1	Cost	7
2.2	Where the money goes	8
2.3	The budget has to fit the model	10
2.4	Seen, and kept	11
2.5	Time and repeated work	11
2.6	Across models	11
2.7	Is pruning needed at all?	12
3	Concluding remarks	13
4	Methods	14
4.1	Finding the opportunity	14
4.2	Role of GPT-6 Astra	14
4.3	Prototype	14
4.4	What did not work	14
4.5	Browser experiment design	14
4.6	How the request is built	16
4.7	The OpenAI Responses adapter	17
4.8	Measurement	17
4.9	Experiment platform	18
4.10	Answer evaluation	18
4.11	Analysis and threats to validity	19
4.12	Reproducibility	19
5	References	20
6	Supplement	21
6.1	Reproducibility details	31
6.2	Earlier experiments	32

0 Technical summary

0.1 Motivation

The aim was to reduce the cost of Asana's StackAI agents on frontier models from two providers, where cache reuse depends on request structure.

0.2 Findings

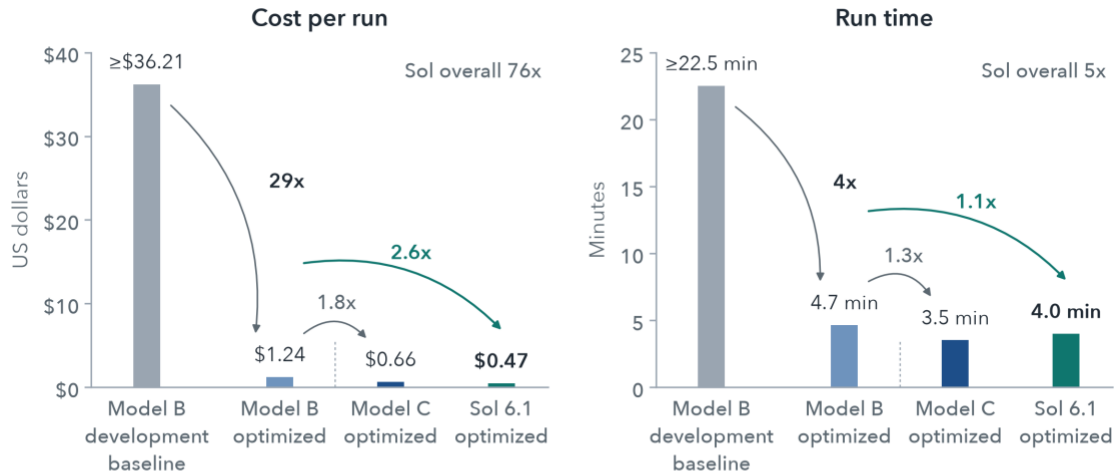
Each request resends the tools, the system prompt and the agent's history of page text and screenshots. The fixed tools and system prompt were cached, but the history was not. Caching the history as well would not have been enough on its own: the agent changed earlier messages at each step, dropping screenshots and trimming text to fit its history budget, and any edit breaks reuse from that point on. The fix changes how often the history changes, not only what is cached: history caching, screenshots pruned in batches, and a budget large enough that old text is not trimmed.

0.3 Test design

The task represents a typical customer workflow: an agent browses books.toscrape.com, a public demo catalog, collects details on 32 books, and returns a table with price totals. The agent's growing history was cached and screenshots were pruned in batches, written N:M, so 20:1 keeps up to 20 screenshots before cutting back to 1: about 19 calls reuse the cached history, then one rewrites it. The history budget, which caps the text and tool calls the agent resends, was also varied; it does not count screenshots, so it controls text trimming, a second source of cache breaks. The study ran in two phases on four models: Model A and Model B, then Model C and OpenAI's GPT-6.1 Sol.

0.4 Results

Caching pays off when history stays stable. On Model B, the benchmark, the best condition cut estimated model cost 29x against the development baseline and ran 4x faster. The same optimized agent on GPT-6.1 Sol cost a further 2.6x less and ran a further 1.1x faster, 76x cheaper overall; on Model C it cost 1.8x less and ran 1.3x faster than on Model B ([Figure 1](#)). On every model, every best-condition run cost less than every baseline run.



Means of 3 runs. ≥: the baseline includes a capped run, so its mean is a lower bound.

The two right folds compare against Model B optimized. Model B ran in phase 1, Model C and Sol 6.1 in phase 2 of the same study (dotted line).

Figure 1. Cost per run and run time of the development baseline on Model B, against the optimized agent on Model B, Model C and Sol 6.1. Means of three runs; the baseline includes a capped run, so it and the folds are lower bounds. Model B ran in the first phase, Model C and Sol 6.1 in the second.

Three findings stand out:

- **Less repeated work.** With the small budget, Model A re-read books whose details were trimmed from history. The larger budget and larger batch-pruning policies removed the re-reads a text-based audit detected.
- The budget has to fit the model. Newer models filled a small history budget faster: Model C first trimmed its history at call 10, against call 64 for Model A. At the small budget, Model C delivered an answer in none of 18 runs and Sol in 3, most stopped by the step limit; at the larger budget, every run on both models answered.
- Caching alone is not enough. At the larger budget, caching the history without batch pruning cost more than not caching it on Model A, Model C and Sol 6.1, and on Model B it worked in only one of three runs at each budget (Section 2.1).

0.5 Scope and limits

- One agent design and one task, with a fixed prompt and architecture, on four models from two providers. Results may differ with other designs or tasks.
- Answers. Every best-condition run on every model saw all 192 facts. Final-answer tuning was outside the study's scope.

1 Introduction

Asana serves more than 170,000 customers ([Asana results](#)) and [acquired StackAI](#) to add cross-system execution for human-agent teams. Its platform handles about a thousand times more web requests than StackAI does today. As Asana brings agent automations to that platform, the cost of browser execution at this scale is an engineering concern. This study examines that concern in a development workflow on a public catalog, ahead of any claim about production savings in Asana.

StackAI is an enterprise platform for building AI agents without code. It supports browser and computer use, agent-to-agent orchestration, evaluations, more than 500 integrations and enterprise governance ([Figure 2](#)).

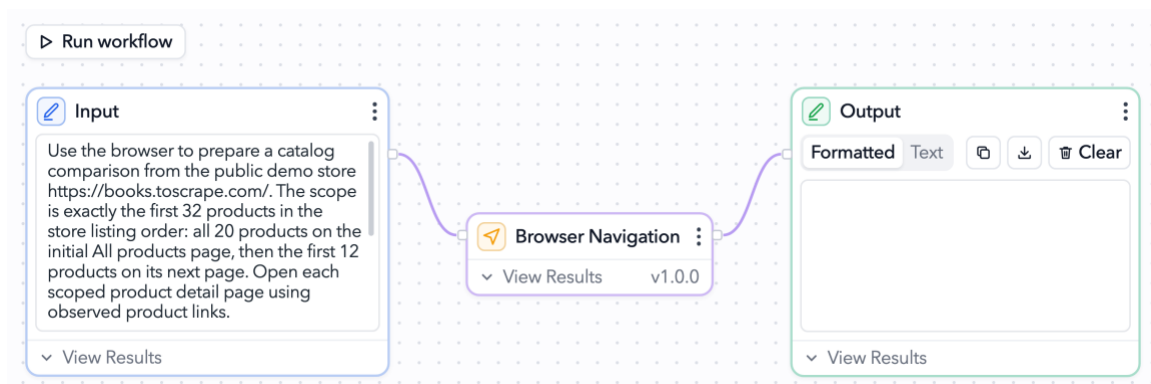


Figure 2. The StackAI workflow for this study: an Input node containing the task prompt, a Browser Navigation node, and an Output node.

Working in [Codex](#), GPT-6 Astra traced a cache-reuse problem to the browser agent's history management: screenshot removal and text trimming changed earlier history. Under human direction, Astra implemented caching and retention controls, ran 144 experimental runs on four models and analyzed request captures and provider usage. The findings connected history management to model cost and repeated browsing work ([Figure 3](#)).

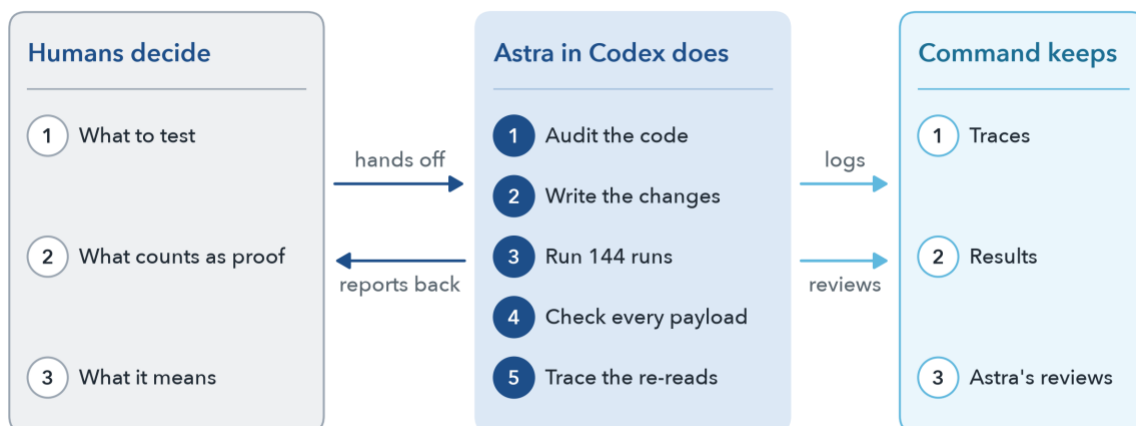


Figure 3. Roles in the study: humans and GPT-6 Astra working in Codex, with Command as the shared record.

ML engineers have long tuned models by searching a hyperparameter space. Agent configuration can be approached the same way: choose settings θ (here, caching policy and history budget) to minimize expected cost per run $C(\theta)$ while still completing the task, $Q(\theta) \geq Q_0$. This study takes a first, manual step toward that search: a grid of twelve settings per model, with quality monitored but not enforced, and a way to identify what prevents unattended searches.

1.1 Prompt caching in brief

On the other provider, a cache write costs 1.25x the normal input price and reading it costs 0.1x. A write pays off on its second use: one write and one read cost 1.35x, against 2x for two uncached sends, and an unread write costs 25% more. A cached prefix can be reused only up to its first change, and a marker can move forward between calls and still hit the previous call's entry. Automatic caching places the moving marker, but neither it nor a hand-placed marker preserves reuse after earlier messages change. OpenAI's Responses API follows the same prefix rule with explicit breakpoints; in this study Sol's cache writes cost 1.25x the input price and reads 0.05x, with a 30-minute lifetime (Table 2).

1.2 Why the cache misses in agents

Browser agents strain this rule. At each step they add a screenshot and page text, and, to stay within limits, drop older screenshots and trim older text. Each edit changes an earlier part of the request. Removing a screenshot alone is enough: adding or removing an image anywhere invalidates the message cache. Reuse after that point is lost; only the tools and system prompt at the front remain reusable. Batch pruning lets screenshots accumulate, then removes many at once, so consecutive requests share a long unchanged prefix between cuts.

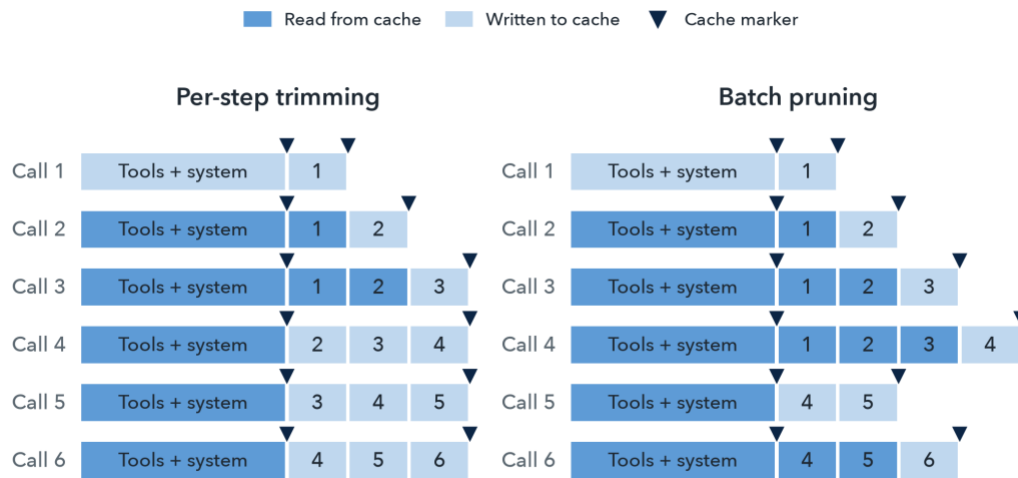


Figure 4. Per-step trimming repeatedly breaks history reuse, while batch pruning preserves it between cuts. Each numbered block represents one agent step, and removing its screenshot alone changes the prefix; the tools and system prompt stay reusable.

2 Results and discussion

Six caching and history-retention policies were tested at history budgets of 120,000 and 480,000 characters, with three runs per condition on four models: Model A and Model B in the first phase, and Model C and GPT-6.1 Sol in the second ([Table 1](#) in Methods). The main text shows four conditions per model, each adding one change to the previous condition; the supplement shows all twelve ([Figures S1 to S6](#)):

- **Development baseline.** Existing code: caching only the fixed system prompt and tools, with a 120,000-character budget.
- **Larger budget.** The same caching policy with the 480,000-character budget.
- **History caching only.** The larger budget plus caching for the agent's growing history, still removing screenshots one at a time.
- **Best condition.** The larger budget plus history caching and batch pruning, 20:10 on Model B and 20:1 on the other three models, the tested condition with the lowest observed mean cost for each model.

2.1 Cost

Batch pruning at the larger budget was the cheapest condition on every model: \$0.84 per run on Model A, \$1.24 on Model B, \$0.66 on Model C and \$0.47 on Sol 6.1. Every best-condition run was cheaper than every baseline run ([Figure 5](#)). Against the development baseline, cost fell 8x on Model A and 29x on Model B. The development baseline runs on Model C and Sol hit the step limit, so for those models the fair comparison holds the budget fixed: against the baseline policy at 480,000 characters, batch pruning cut cost 2x on Model C and 4x on Sol. These savings did not result from skipping books: every best-condition run saw all 192 facts (Section 4.10).

History caching only was erratic on Model B, reading about half its input from cache in one of three runs at each budget ([Figure S2](#)). At the large budget, the split reflected its tool use: the high-read run often read page text separately, leaving history unchanged, while in the low-read runs almost every request removed an earlier screenshot and reused only the tools and system prompt (Section 4.6).

On the other three models, history caching only cost more than no history caching at the larger budget, because cache writes made up over 90% of its cost while cache reads stayed below a third of input ([Figure S2](#)).

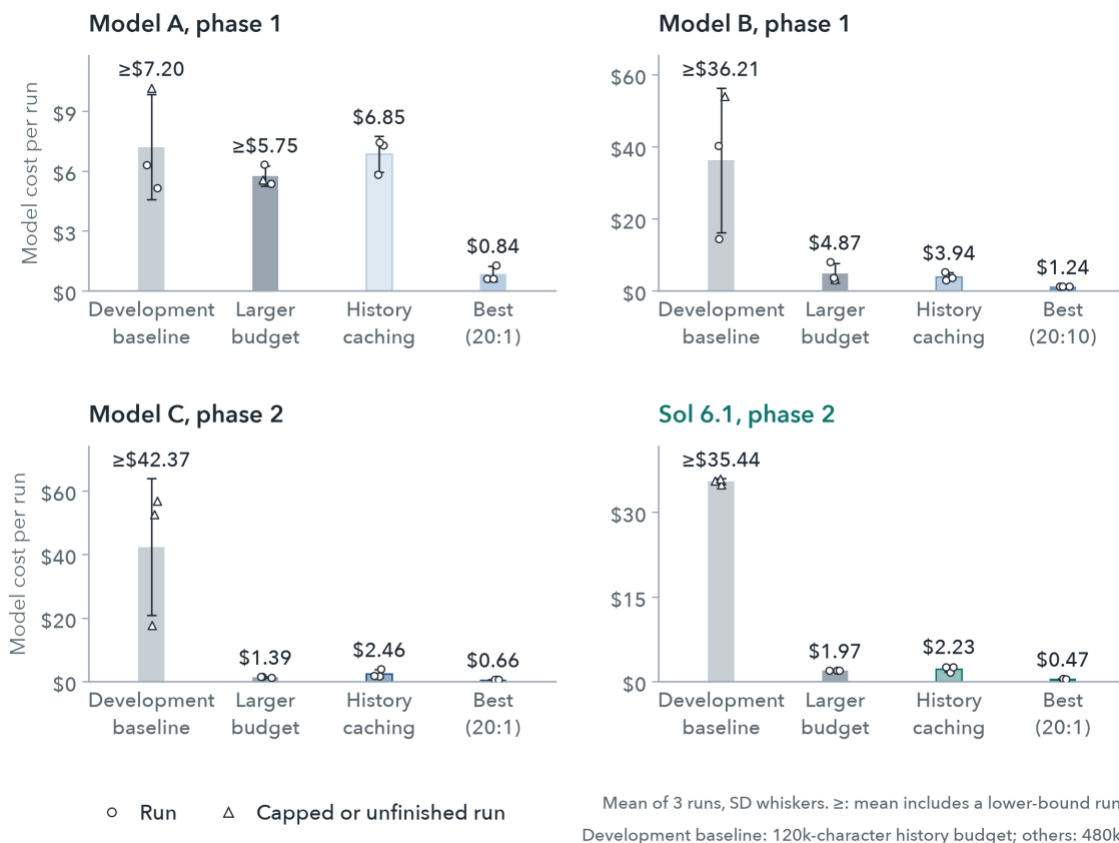
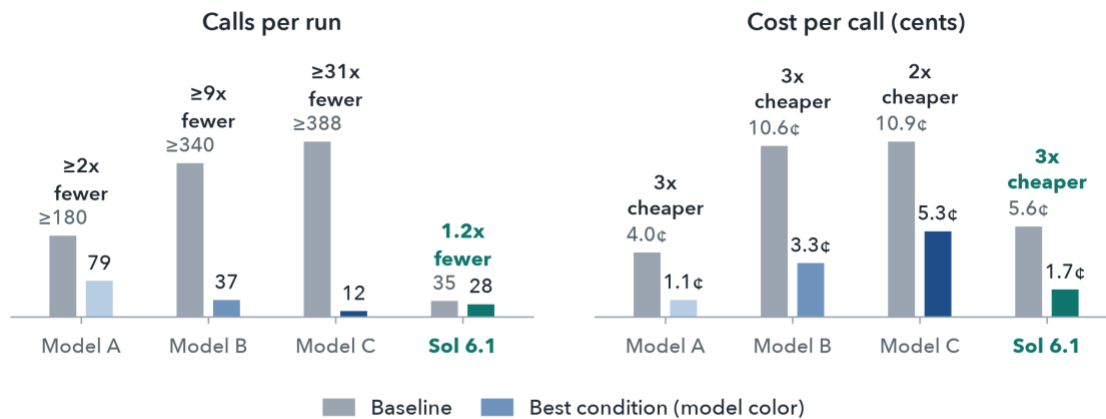


Figure 5. Model cost per run for the four main conditions on each model. Triangles and ≥ mark means that include capped or unfinished runs, so they are lower bounds. [Table S2](#) lists every run.

2.2 Where the money goes

Cost per run equals calls per run times cost per call, and both fell ([Figure 6](#)). On Models A, B and C, the larger budget cut calls and batch pruning made each call cheaper: Model B made 9x fewer calls and Model C 31x fewer, and cost per call fell 2x to 3x. Against its own baseline at the same budget, Sol made 1.2x fewer calls and each call cost 3x less. Calls cost less because input moved from full price to cache reads ([Figure S7](#)). In the best condition, cache reads made up 94% of input on Model A, 88% on Model B, 73% on Model C and 89% on Sol.



≥: the mean includes a capped or unfinished run, so the count and fold are lower bounds.
 Baseline: the 120k development baseline, except Sol, whose 120k runs all hit the step cap; Sol uses its 480k baseline.

Figure 6. The two factors of cost per run, calls per run and average cost per call, for each model's baseline and best condition. Cost per call is mean cost divided by mean calls. The baseline is the development baseline, except for Sol, whose development baseline runs all hit the step limit; Sol is compared with its baseline at 480,000 characters. Ratios are rounded down.

The savings came despite a larger context. Best-condition calls carried more input because the agent kept more history and screenshots: on Model B, call 30 averaged 93,000 tokens against 59,000 under the development baseline. The agent read most of that from cache at 0.1x the price and made far fewer calls, so each run sent fewer tokens overall: 2.3 million tokens on Model B against 19.7 million, and 4.5 million on Model A against 7.2 million.

Sol's median-cost runs at the larger budget show the same pattern ([Figure 7](#)). In the baseline-policy run, call 24 carried about 40,000 input tokens, 9% of them from cache, and the run took 35 calls. In the best-condition run, call 24 carried about 58,000 tokens, 93% from cache, and the run finished in 24 calls. Sol reads cached input at 0.05x the input price, so the larger, mostly cached request cost less than the smaller uncached one.

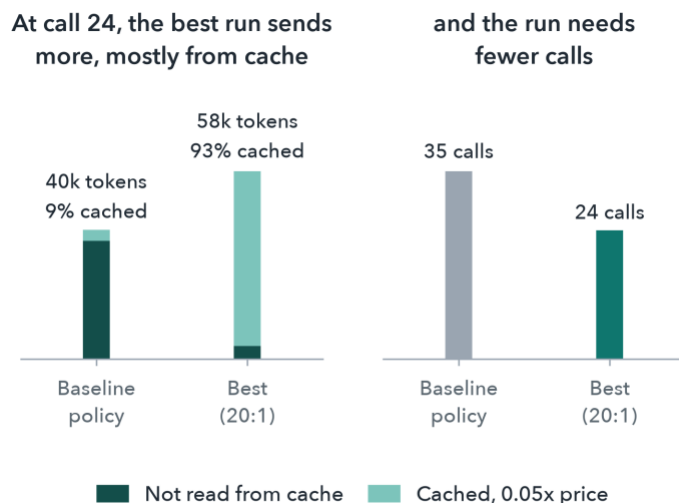
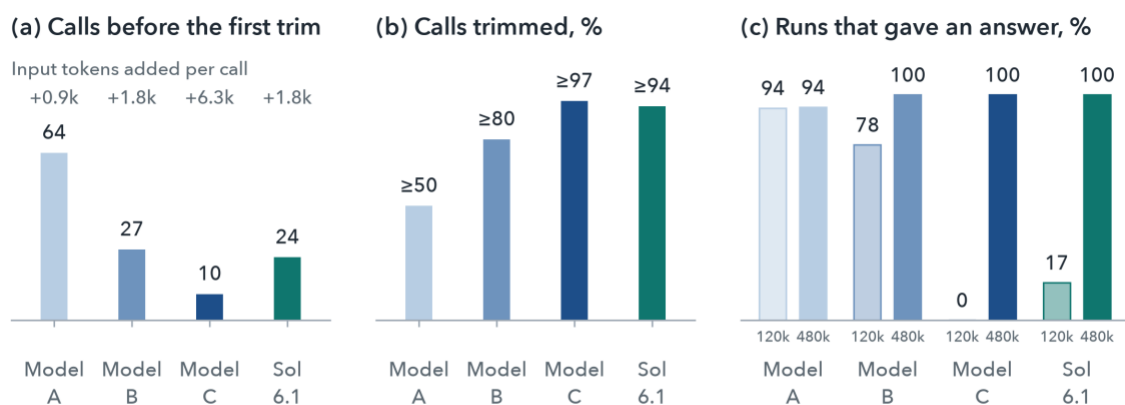


Figure 7. A closer look at the median-cost Sol 6.1 runs under the baseline policy and under the best condition, both at 480,000 characters. Input and cached share refer to call 24, the last call both runs share.

2.3 The budget has to fit the model

Newer models made the small budget more damaging (Figure 8). Model C added about 6,300 input tokens per call before its first trim, against 900 for Model A and 1,800 for Model B and Sol, and first trimmed its history at call 10, against call 64 for Model A. At 120,000 characters, 97% of Model C's calls and 94% of Sol's trimmed text, against about half of Model A's, and the two newer models delivered an answer in 0 and 3 of 18 runs; most stopped at the step limit. At 480,000 characters no call on any model trimmed text, and every Model C and Sol run delivered an answer. On Model B, two batch-pruning conditions cost more on average than the baseline at the small budget (Table S4).



(a) and (b) are at the 120k history budget; no call was trimmed at 480k. (c) compares the 120k and 480k budgets.

≥: includes runs that hit a cap or ended early, so the true value is at least this high.

Figure 8. The small budget on each model: (a) calls before the first text trim, with the input tokens each call added before it, and (b) the share of calls that trimmed text, both at 120,000 characters; (c) the share of runs that gave an answer at each budget. ≥ marks lower bounds from capped or unfinished runs.

2.4 Seen, and kept

Seeing every fact is not the same as keeping it (Figure 9). At 120,000 characters, nearly every run on every model saw all 192 facts at some point, but trimming removed some of them again, and no final request still held all 192. At 480,000 characters, the final request held all 192 in 14 of 18 runs on Model A, 15 on Model B, 17 on Model C and 18 on Sol 6.1. A run that has lost facts from its history must read them again or answer without them, one source of the small budget's repeated work and missing answers.

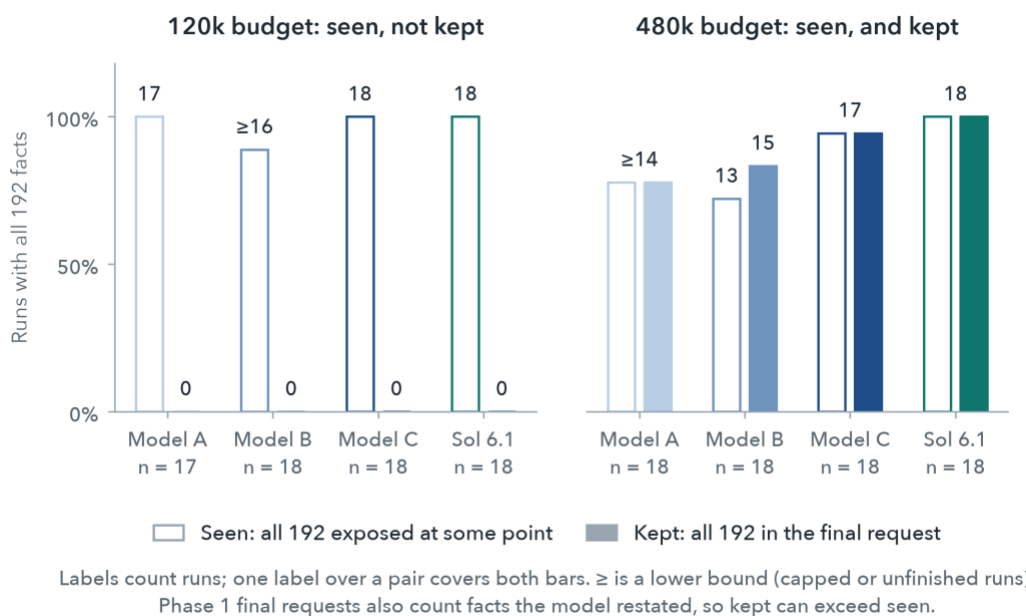


Figure 9. Seen and kept, by model and budget: runs that saw all 192 facts at some point (outline) and runs whose final request still held all 192 (filled). $\geq$ marks lower bounds. Phase 1 final requests also count facts the model restated, so kept can exceed seen.

2.5 Time and repeated work

Mean run time fell from 27 minutes to under 5 on Model A and from 22 minutes to under 5 on Model B. The best conditions on Model C and Sol took 3.5 and 4.0 minutes (Figure S8).

The small budget led to some repeated work. On Model A, a text-based audit by Astra found 130 re-reads of products whose details were no longer in the request, 93 in two runs, and the larger budget and the larger batch-pruning policies, 10:1 and up, removed every re-read it detected (Figure S6). On Model B, page revisits fell from at least 271 per baseline run to 2 under the larger budget.

2.6 Across models

The comparison across models is descriptive, because the two phases used different request settings (Section 4.5). On this task, the optimized agent was cheapest on Sol, at \$0.47 per run, 2.6x less than on Model B and 1.3x less than on Model C. It ran in 4.0 minutes on Sol, against 4.7 on Model B and 3.5 on Model C. Sol made more calls than Model C, 28 against 12, but read 89% of its input from cache at \$0.10 per million tokens, against \$0.20 on Models B and C.

2.7 Is pruning needed at all?

Not on this task. A follow-up kept every screenshot, so the history was only ever appended to (Section 4.5). Cost per call was 1.2x lower than in the best condition on Model B and Sol and about 5% lower on Model C, and every run stated the correct total ([Table S7](#)). Pruning still matters for long tasks, for models with smaller context limits and for providers with more expensive cache reads.

Cost per run is noisier than cost per call, because the number of calls varies from run to run: a model sometimes visits two book pages in one call, and the three Sol runs at 20:1 took 24, 25 and 35 calls. With three or four runs per condition, the study maps the landscape rather than ranking conditions a few percent apart, which would need many more runs.

3 Concluding remarks

Cache markers and history trimming need to be designed together. On this browser workflow, keeping cached history stable for longer and matching the agent's history budget to the model cut estimated model cost 29x on Model B, and the same optimized agent on GPT-6.1 Sol cost \$0.47 a run, 76x less than the original setup. The lessons held on both providers. For agents using a provider that caches by prefix, such as the two studied here:

- Treat the agent's history as an append-only log where possible.
- Put a cache marker at the end of the newest tool result, or use automatic caching, and keep an explicit marker on the system prompt, which is still read after a prune. Expect an edit to an earlier message to break reuse from that point onward.
- Avoid editing earlier history when the context allows; when pruning is needed, prune in large batches and measure the trade-off.
- Size the history budget to the model, and check how often history is trimmed and whether the agent repeats work.
- Measure cache reads per call, and verify the payload received by the provider.

Asana is applying these findings in StackAI to improve computer and browser navigation for automation at Asana's scale. Planned platform changes will help teams use GPT-6 Astra in Codex to run experiments faster and compare workflow configurations. The longer-term vision is to bring this experimentation into the platform's own evaluations, enabling Asana's customers and internal teams to optimize cost and runtime alongside answer quality. This study illustrates the role Astra and Codex could play: connecting code investigation, implementation and measured results to make agent optimization a repeatable platform capability. The caching and retention changes have shipped in StackAI.

4 Methods

4.1 Finding the opportunity

[GPT-6 Astra](#) audited the StackAI codebase and its three routes to the other provider's models: the direct API and two routes through a cloud host's inference service. Before behavior was changed, requests and responses were instrumented to link every provider receipt to its run. The audit pointed to browser navigation, which edited its own earlier history ([Figure 4](#)). The first phase used that provider's direct API; the second also used OpenAI's Responses API for Sol, through a new adapter (Section 4.7). [Table 2](#) lists the prices.

4.2 Role of GPT-6 Astra

Humans directed the work and reviewed its conclusions ([Figure 3](#)). GPT-6 Astra in Codex investigated the code, implemented experiment controls, built the workflow harness and analyzed the recorded results in both phases.

The harness dispatched workflows in waves of twelve and recorded provider usage, requests, settings, browser actions, eviction events and outputs. Settings were traced from the workflow through the sandbox to outgoing requests and provider receipts. Evidence was kept in Command.

Configuration mismatches and capture failures required fixes, and some historical evidence remained incomplete. Human review challenged results and settings, while [a separate model](#) independently reviewed plans, code and reporting. Astra subsequently reconciled costs and traces, assembled the run-level spreadsheets and wrote the retrospective analyses.

4.3 Prototype

The mechanism was first tested on a regular tool-calling agent reading three documents in sequence, with each call carrying prior results. Five route and model combinations were tested. Caching the growing tool history cut estimated input cost by about 31% on every combination. A one-call negative control that wrote the cache but never reused it cost about 25% more. The same lesson applied to the browser work: a cache write pays only if a later call reads it.

4.4 What did not work

- Keeping more screenshots without extending the cache boundary just added uncached input.
- Some early browser trials raised cache reuse but still increased whole-run cost or made answer delivery worse.

4.5 Browser experiment design

The task, named CAT32 for its 32 catalog books, is a StackAI workflow whose browser agent opens books.toscrape.com, a public demo catalog, visits 32 book pages, collects six fields per book (title, URL, UPC, price, availability and stock quantity), and returns one table with the price total and the cheapest and most expensive books ([Figure 2](#)). The reference answer was built independently from the product pages, with a total of £1,164.91, and a later check confirmed the same prices. The site was live, so the reference cannot guarantee identical pages across runs. Each

model call runs one sequence of browser actions, so a model may visit more than one book page per call; call counts therefore reflect both the caching policy and how the model batches its work.

[Table 1](#) summarizes the design, which ran in two phases: Model A and Model B from 14 to 23 September 2026, then Model C and GPT-6.1 Sol on 30 September 2026. Each wave ran every condition once, spreading wave-level effects such as time of day and provider load evenly across conditions. Task failures remained in the results ([Table S1](#)). One Model A run failed technically after six calls; it is excluded from the plotted means, so 20:10 at the small budget has two runs. The original ledger retains it, and [Table S3](#) shows results with and without it.

Table 1. Experiment design.

	Phase 1	Phase 2
Models	Model A, Model B	Model C, GPT-6.1 Sol
What they are	Model A: a smaller, lower-cost model from another frontier lab, released Fall 2025. Model B: the model originally used in production, from the same lab as Model A, released Summer 2026.	Model C: a newer version of Model B, released Fall 2026. GPT-6.1 Sol: OpenAI's model.
API	Other provider's messages API	Other provider's messages API (Model C), OpenAI Responses (Sol)
Dates	14 to 23 Sep 2026	30 Sep 2026
Task	CAT32: 32 books x 6 fields = 192 facts, total £1,164.91	Same
Conditions	6 policies x 2 budgets (120k, 480k characters)	Same
Runs	3 per condition, 72; 71 analyzed	3 per condition, 72; 72 analyzed
Scheduling	Waves of 12 parallel runs	Blocks of up to 12, by replicate
Limits	500 steps, 16,384 output tokens, \$20 (Model A) or \$75	Same, \$75
Cache lifetime	5 min	5 min (Model C), 30 min (Sol)
Effort	Unset, model defaults	High; reasoning replayed for retained turns
Tool choice	Read and done forced on some calls	Automatic (Model C), forced (Sol)
Cache markers	System prompt, plus newest tool result	Same; Sol keeps one earlier matching breakpoint
Fact exposure	Tool text and screenshot audit	Fields in outgoing request text
Answer scoring	Audit against the reference	Two independent model reviewers

In the first phase, the request code left thinking, effort and temperature unset, so model defaults applied. Model B defaults to high effort and decides when to think, which it did on about one call in seven in the baseline runs, while Model A does not think unless asked. Thinking settings stayed constant within each run and did not invalidate the cache. The character budgets equal roughly 30,000 and 120,000 tokens of text, and [Table S6](#) lists each model's actual input tokens per request.

In the second phase, effort was set explicitly to high on both models. Model C used adaptive thinking, with its signed thinking blocks replayed, and chose its tools itself instead of the forced read or done tool. Sol ran through the OpenAI Responses API with high reasoning effort, the standard service tier, no server-side conversation state and encrypted reasoning replayed for retained turns, with images sent at high detail. Runs were capped at 500 steps, 16,384 output tokens and \$75.

A follow-up on 1 October 2026 (UTC) ran 12 runs at once, four each on Model B, Model C and Sol, at 480,000 characters with history caching and no screenshot pruning, using a cap of 100 screenshots that no run reached. It used the second phase's build and settings; Model B kept its first-phase default effort. It was prompted by an estimate from the recorded calls of a 35-call Sol run at 20:1, which put its cost without the prune at \$0.49 instead of \$0.51; the three 35-call Sol runs without pruning cost \$0.49 each. No run trimmed text, the largest request held about 151,000 input tokens, and 11 of 12 runs saw all 192 facts. Model A was left out because one earlier Model A run's final request, with its screenshots restored, exceeded its 200,000-token input limit. A prune pays for itself only once the reads it saves exceed the cost of rewriting the history: on the 35-call Sol run, after about 27 more calls at Sol's cache-read price of 0.05x the input price, and after about 13 at the other models' 0.1x.

4.6 How the request is built

The development baseline places one cache marker on the system prompt, also covering the tool definitions the other provider reads first. History caching adds a second marker on the last block of the newest tool-result message, moving forward on every call. It still hits because the other provider looks back up to 20 blocks from the marker for an earlier cache entry, and consecutive tool calls or tool results count as one block, so each agent step adds only a few. Changing tool choice also resets the message cache. The agent forced its read or done tool on some calls, and each such call caused one miss before reuse resumed. A marker only allows prefix reuse, and the next response's cache-read counter shows whether the prefix was reused.

[Figure 10](#) sketches a simplified request from the best Model B condition, with some content omitted. One cache breakpoint sits on the system prompt and the second on the newest tool-output block, not on its nested screenshot.

```
{
  "model": "model-b",
  "system": [{ "type": "text", "text": "...",
    "cache_breakpoint": true }],
  "tools": ["... 16 tool definitions ..."],
  "messages": [
    { "role": "user", "content": [{ "type": "text", "text": "... task ..." } ] },
    { "role": "assistant", "content": [{ "type": "tool_call", "name": "navigate", ... } ] },
    { "role": "user", "content": [{ "type": "tool_output",
      "content": [{ "type": "text", "text": "... page text ..." },
        { "type": "image", "source": "... screenshot ..." } ],
      "cache_breakpoint": true } ] }
  ]
}
```

Figure 10. A simplified request from the best Model B condition, with some content omitted, field names generalized and the two cache breakpoints highlighted.

Tool results carry screenshots. The baseline keeps only the newest one and removes older images without placeholders, preserving the surrounding text. Batch-pruning policy N:M allows up to N screenshots. When the count exceeds N, it removes older, previously sent screenshots down to M.

New screenshots are protected, so the count can briefly exceed the target, and removed screenshots are never restored.

The history budget counts message text and tool calls, but not images, the system prompt or tool definitions. It keeps the original task and the latest complete steps. The limit is soft because the newest step can exceed it. Text trimming runs before screenshot selection.

No run at 480,000 characters reached that limit: no call trimmed text, in the main study or the follow-up, so on this task the larger budget acted as no limit at all. It could be removed here, but not safely in general. An agent that drifts, for example by looping or revisiting pages, keeps growing its history toward the model's context limit, and anything that then breaks the cache makes every call pay full price for that context. A harness therefore needs several guardrails, such as limits on steps, tokens and cost per run, so that a run that goes wrong stops before it becomes expensive. Compaction, which summarizes older history instead of dropping it, is another way to keep long runs within bounds, but it is outside the scope of this study.

Comparing archived requests explains the history-caching-only split on Model B at the large budget. Ignoring cache-marker fields, the high-read run had 47 requests that only appended to the previous one, out of 91 transitions, against 1 and 3 in the two low-read runs. In all three runs, the other transitions removed an earlier tool result's image but kept its text, and all 102 of those read only the 6,228-token tools and system prompt. The high-read run often read page text separately without a screenshot, leaving history intact, while the low-read runs mostly combined navigation and reading in calls that produced screenshots. The comparison does not explain why the agent chose those paths or account for the small-budget runs.

4.7 The OpenAI Responses adapter

Sol required an adapter for the Responses API rather than the other provider's payloads with a renamed model. It sent the retained context explicitly on every call, kept opaque reasoning items with the turns they belonged to, and never let evicted turns return through hidden server state. Caching used explicit breakpoints with a 30-minute lifetime: the baseline placed one on the system and developer prefix, and history caching added one on the newest function-call output. Before the formal runs, a six-call smoke test showed reuse lost at a screenshot prune, so the history conditions also kept at most one earlier breakpoint, and only while its full prefix still matched. The API reported cache writes and reads natively.

4.8 Measurement

Each call was measured using the provider's native token counters: uncached input U, cache writes W, cache reads R and output O. Total input is $U + W + R$. Cache-read share is R divided by total input, calculated by summing tokens within each run, then averaged across runs. Model cost is $(U \times \text{input price} + W \times \text{write price} + R \times \text{read price} + O \times \text{output price})$ divided by one million, using the prices in [Table 2](#). These estimated model costs are not invoices and exclude sandbox and server costs. Missing usage is recorded as unknown, never zero. A text-trimming call is a request whose message count fell after trimming. Run time is StackAI's workflow time. The five-minute cache lifetime was sufficient: consecutive calls were at most 88 seconds apart on Model A and 69 seconds on Model B, and each cache read refreshes the lifetime. Model C used

the same five-minute lifetime and Sol a 30-minute one. No request exceeded about 151,000 input tokens, below the 270,000-token threshold above which Sol's long-context prices apply, so standard prices held throughout.

Table 2. Prices used in the analysis, in US dollars per million tokens. Phase 1 prices are the other provider's list prices; phase 2 prices were frozen on 29 September 2026 from both providers' pricing.

Model	Input	Cache write	Cache read	Output	Cache lifetime
Model A	\$1.00	\$1.25	\$0.10	\$5.00	5 min
Model B	\$2.00	\$2.50	\$0.20	\$10.00	5 min
Model C	\$2.00	\$2.50	\$0.20	\$10.00	5 min
Sol 6.1	\$2.00	\$2.50	\$0.10	\$10.00	30 min

4.9 Experiment platform

The tested settings were exposed through StackAI's workflow UI and API on an experimental branch. Early phases ran one app stack per condition. The final cohorts shared one local frontend and backend with Redis ([Figure 11](#)). Astra's harness dispatched full workflows with run-specific settings, and its checks traced each setting through the sandbox and the outgoing request to the provider's receipt, to verify what was sent rather than rely on configuration labels.

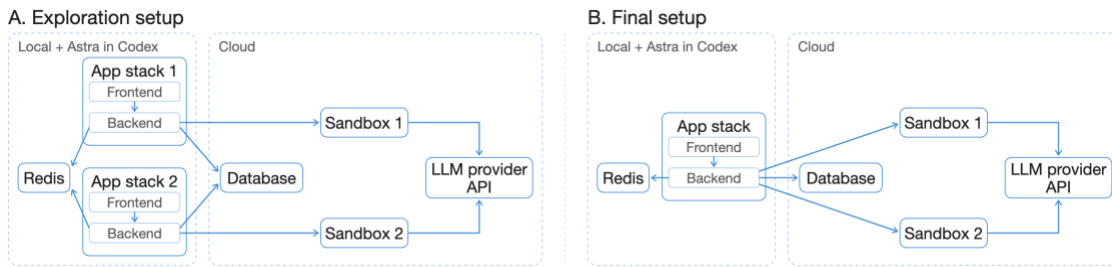


Figure 11. Experiment environments. (A) Exploration: a separate app stack per condition. (B) Final: one app stack, with settings sent to each run's sandbox.

4.10 Answer evaluation

Persisted workflow outputs were scored against the reference answer, rather than the agent's claim of success.

Quality was judged first on retrieval: whether a run saw all 192 reference facts in the tool text it received, and whether its final request still held them ([Table 3](#)). Every best-condition run on every model saw all 192. Answers were also scored on the price total. Model B stated the correct total, £1,164.91, in 30 of 36 runs, and Model C and Sol in every run that delivered an answer; Sol answered with a short summary of the total and the cheapest and most expensive books rather than the full table. Model A found the prices but often summed them incorrectly, stating the correct total in only 2 of 35 runs. The prompt and tools were not tuned for the final answer.

Table 3. Retrieval and answers by model and budget: runs that saw all 192 facts, kept them in the final request, delivered an answer and stated the correct price total. Cells count runs. Phase 1 counts of runs that saw all 192 facts include screenshot text; Figure 9 counts tool text only.

Model	Budget	Runs	Capped	Saw 192	Kept 192	Answered	Correct
Model A	120k	17	1	17	0	16	1

Model	Budget	Runs	Capped	Saw 192	Kept 192	Answered	Correct
Model A	480k	18	1	14	14	17	1
Model B	120k	18	4	16	0	14	13
Model B	480k	18	0	14	15	18	17
Model C	120k	18	18	18	0	0	0
Model C	480k	18	0	17	17	18	18
Sol 6.1	120k	18	15	18	0	3	3
Sol 6.1	480k	18	0	18	18	18	18

4.11 Analysis and threats to validity

Results show condition means with run counts, individual runs as dots, and standard deviation as a measure of spread, not a confidence interval. Improvements are relative to the development baseline and rounded down.

- The best condition was selected after the runs for the lowest mean cost among twelve. Consistent results across large-budget batch-pruning conditions support this choice but do not remove selection bias.
- Parallel runs competed for the same machine and provider, limiting run-time comparisons.
- Step, budget and timeout limits cut some runs short; capped baseline runs make the baseline mean a lower bound. Lower spending from an early stop is not necessarily an improvement.
- The study covers one task, one browser agent and four models from two providers, with three runs per condition except for one Model A condition with two.
- The re-read audit is lexical and detects only re-reads it can trace. Model B's revisit count includes routine catalog returns. Neither measures all repeated work.
- The follow-up without pruning ran as one batch after the main study, with four runs per model sharing caches, so its comparison with the main conditions is indicative.
- The two phases differ in protocol. The second set effort explicitly, let Model C choose its tools, and gave Sol a 30-minute cache lifetime and its own breakpoint rule, so comparisons across models are descriptive.
- In the second phase, concurrent runs could share cached prefixes. Reuse across runs stops at the first differing token, so in practice only the shared tools and system prompt could hit.
- Every development baseline run on Model C and Sol hit the step limit, so their baseline means are lower bounds and are not used as the reference for those models' improvements.

4.12 Reproducibility

The final cohorts ran through one local StackAI frontend and backend on a laptop, with the browser agent in remote [E2B](#) sandboxes and all inference through the two providers' APIs. The first phase ran between 14 and 23 September 2026 with requests naming model-a and model-b, and the second on 30 September 2026 with requests naming model-c and gpt-6.1-sol. The supplement records the hardware, request settings and limits needed to reproduce the runs ([Table S5](#)).

5 References

- Asana. Asana acquires StackAI, adding cross-system execution for human-agent teams. [Press release](#).
- Asana. Fourth quarter and fiscal year 2026 results. [Investor release](#).
- Books to Scrape. A fictional bookstore for scraping practice. [Website](#).
- E2B. Sandboxes for AI agents. [Website](#).
- OpenAI. GPT-6 Astra: A new generation of intelligence. [Announcement](#).
- OpenAI. Codex. [Product page](#).
- OpenAI. Prompt caching. [OpenAI API documentation](#).

6 Supplement

The supplement covers all twelve conditions per model, the run ledger, reproducibility details and a brief account of earlier experiments.

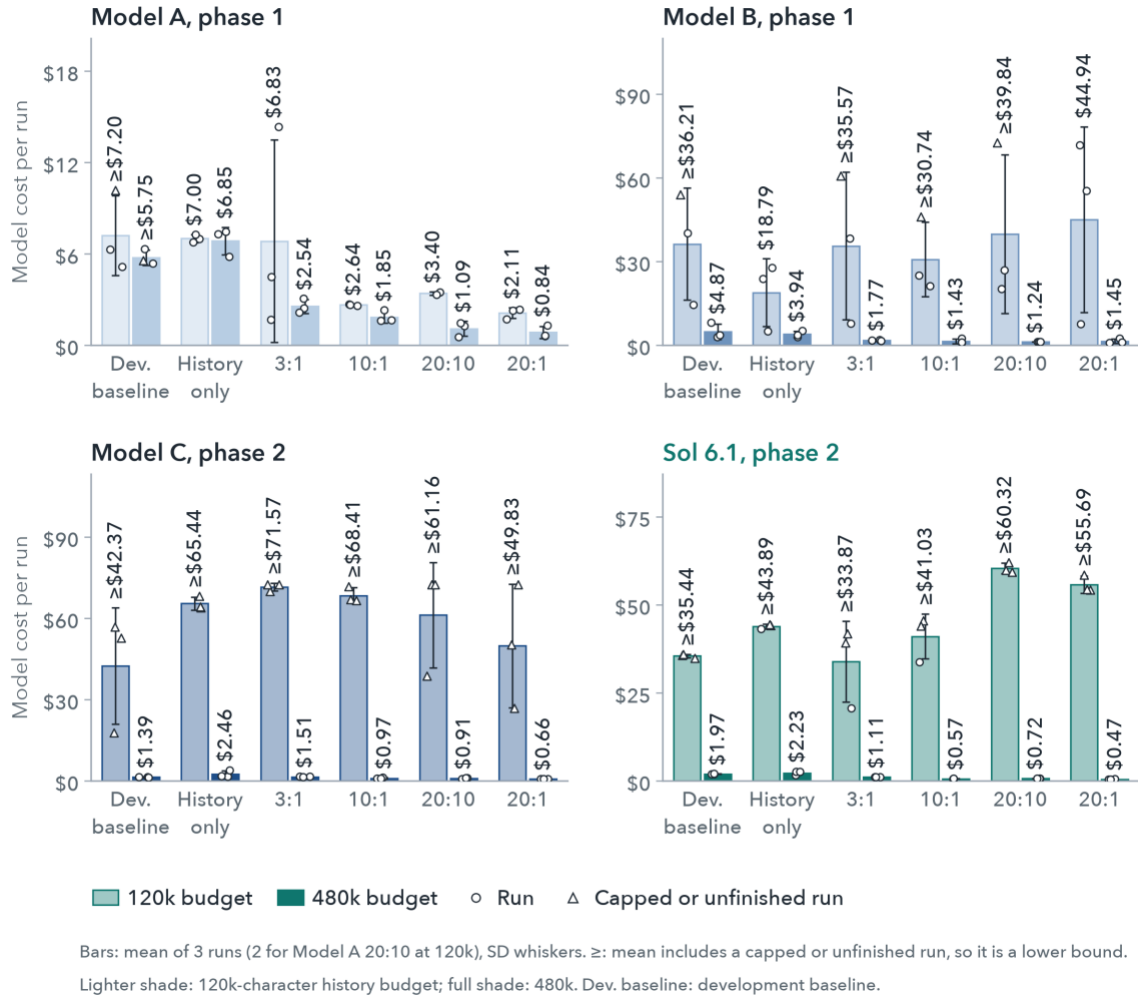
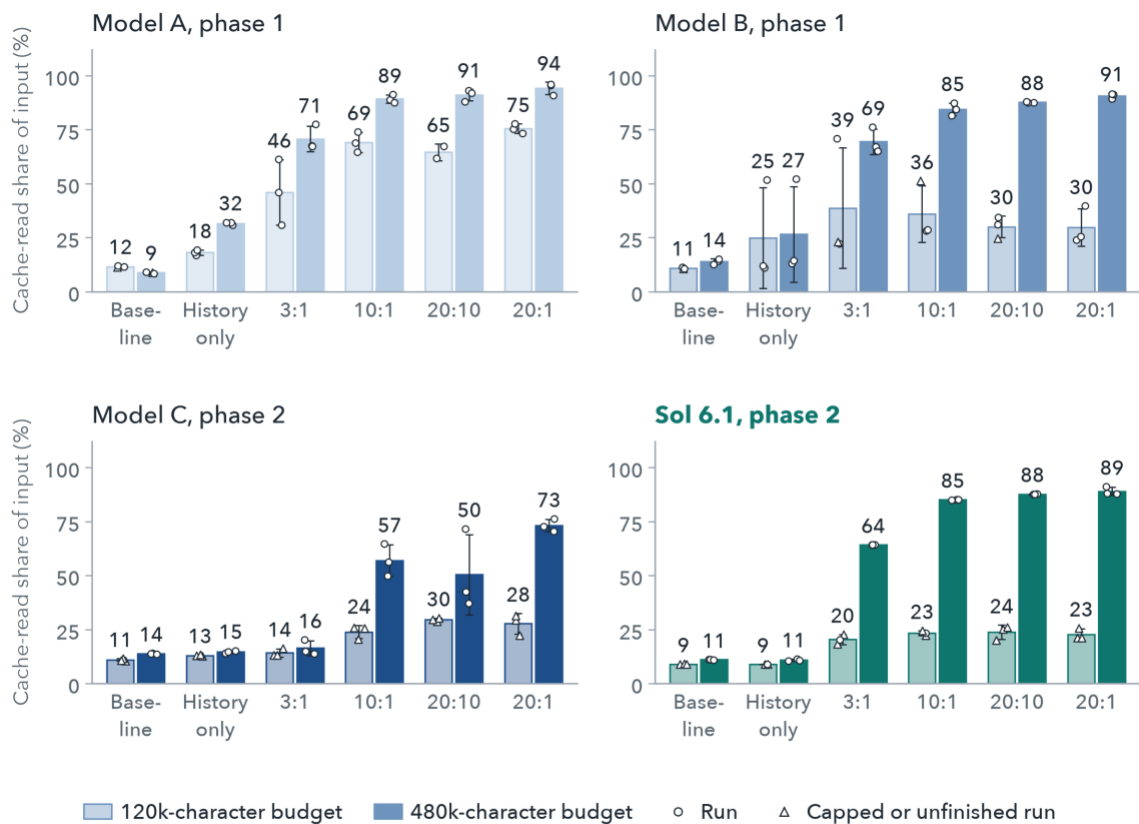


Figure S1. Model cost per run across all twelve conditions for all four models. Triangles and $\geq$ mark means that include capped or unfinished runs, so they are lower bounds.



Mean of 3 runs (Model A 20:10 at 120k: 2, one technical failure dropped), SD whiskers, label = mean %. Lighter bars: 120k.
Share = cache reads / (uncached + cache writes + cache reads), summed per run; a capped run's share covers its receipted calls.

Figure S2. Cache-read share of input across all twelve conditions for all four models.

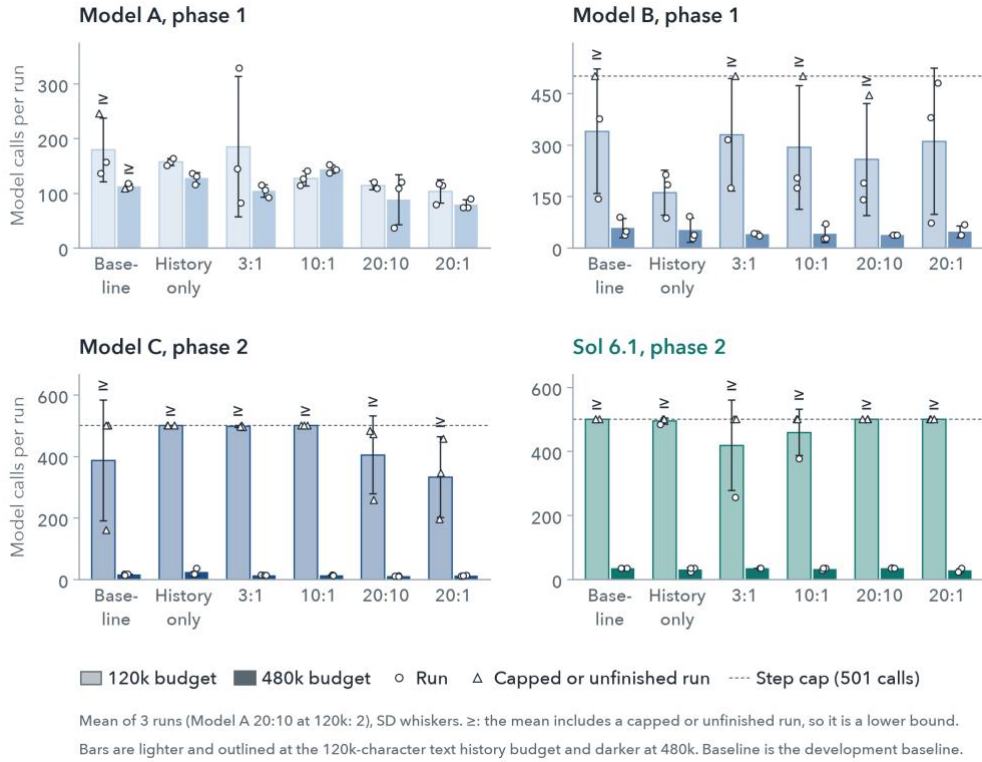
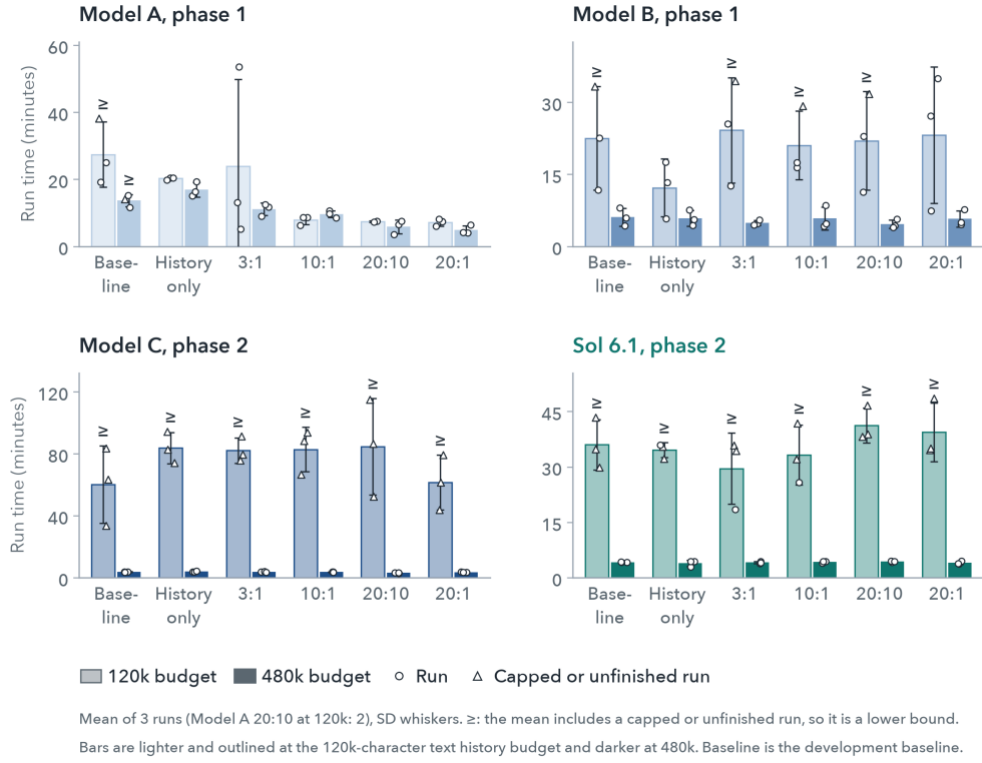


Figure S3. Run time (top) and model calls per run (bottom) across all twelve conditions for all four models. The dashed line marks the step limit.

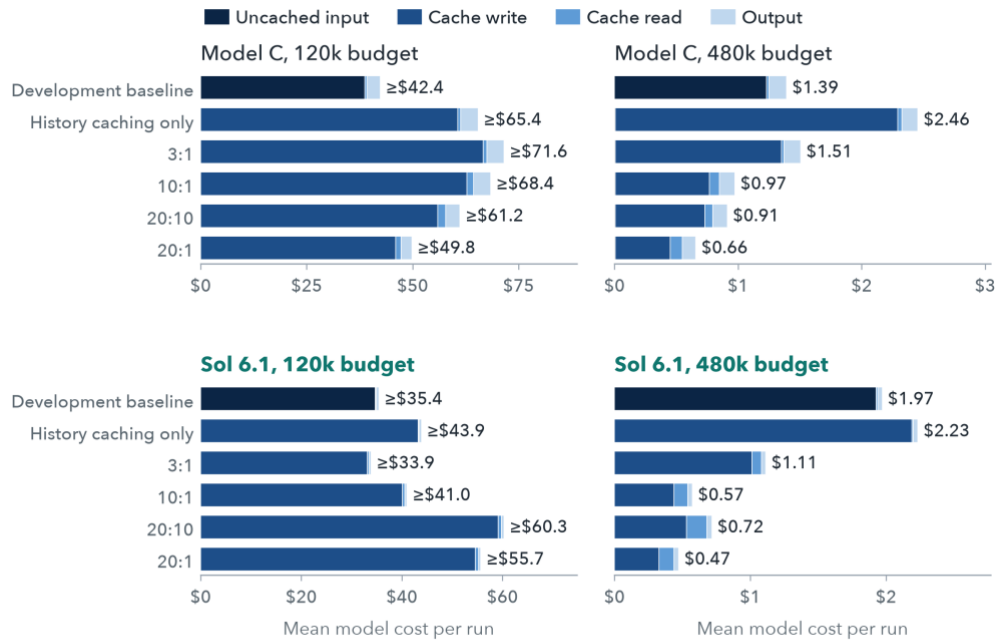
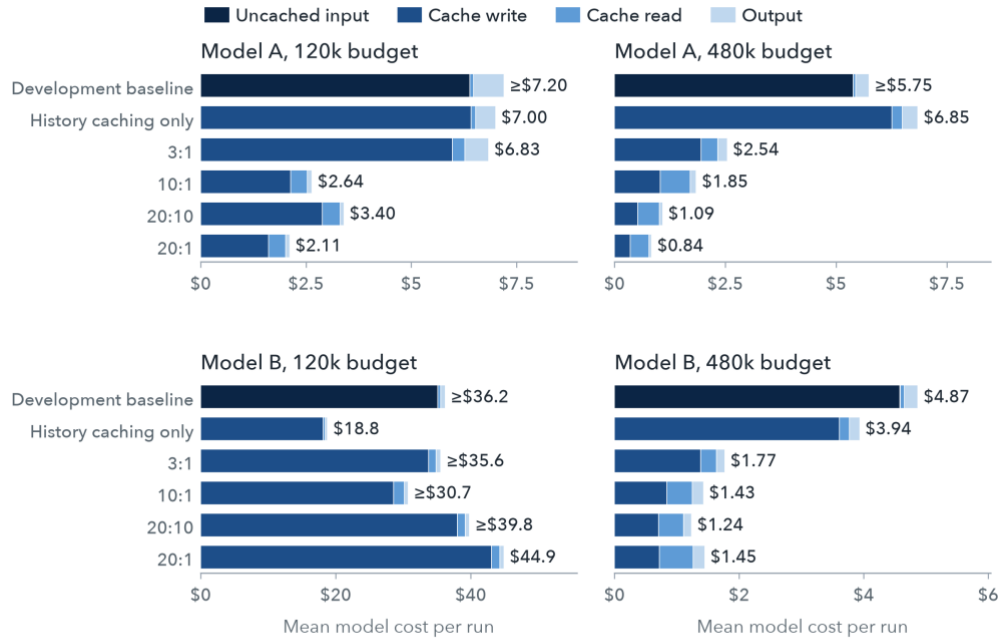
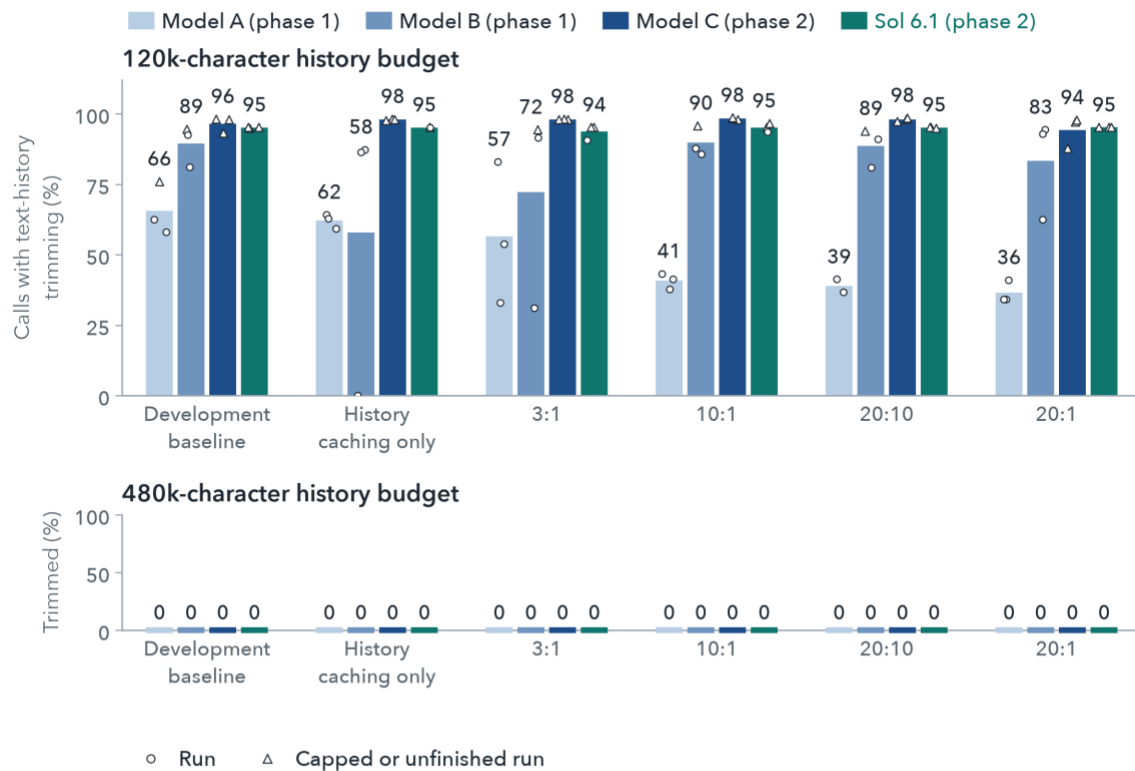
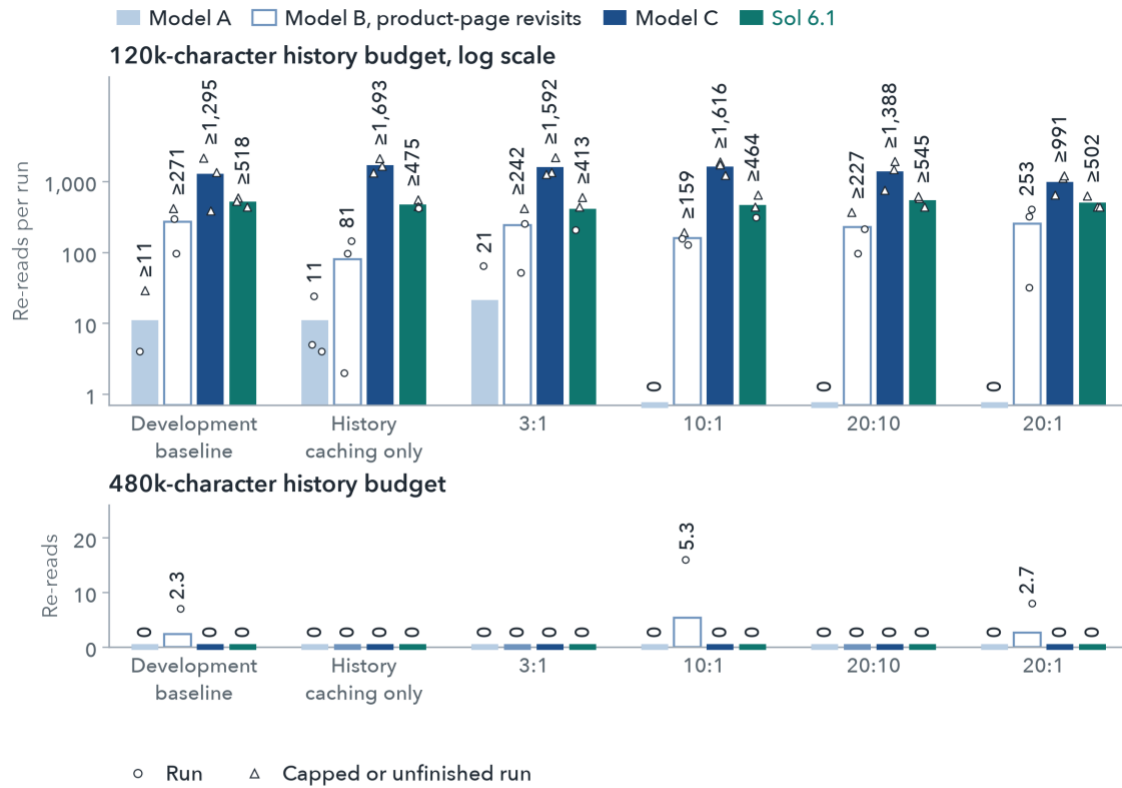


Figure S4. Cost breakdown by token type for all six policies at both budgets, for the first phase (top) and the second (bottom).



Bars: mean of 3 runs (Model A 20:10 at 120k: 2 runs; the failed run is excluded).
 At 480k no request was trimmed in any of the 72 runs (3,656 calls).

Figure S5. Calls with text-history trimming across all twelve conditions for all four models. No call trimmed text at 480,000 characters.



Phase 1: Model A and Model B. Phase 2: Model C and Sol 6.1. Re-read after loss: a product page read again after its UPC had left the preceding request. Model B has no request audit, so its hollow bars count repeated product-page visits (for Model A the two counts differ by at most 2 per run). Bars: mean of 3 runs (Model A 20:10 at 120k: 2). ≥: mean includes a capped or unfinished run, so it is a lower bound.

Figure S6. Repeated work across all twelve conditions: re-reads after loss for Model A, Model C and Sol 6.1, and product-page revisits for Model B, on a log scale.

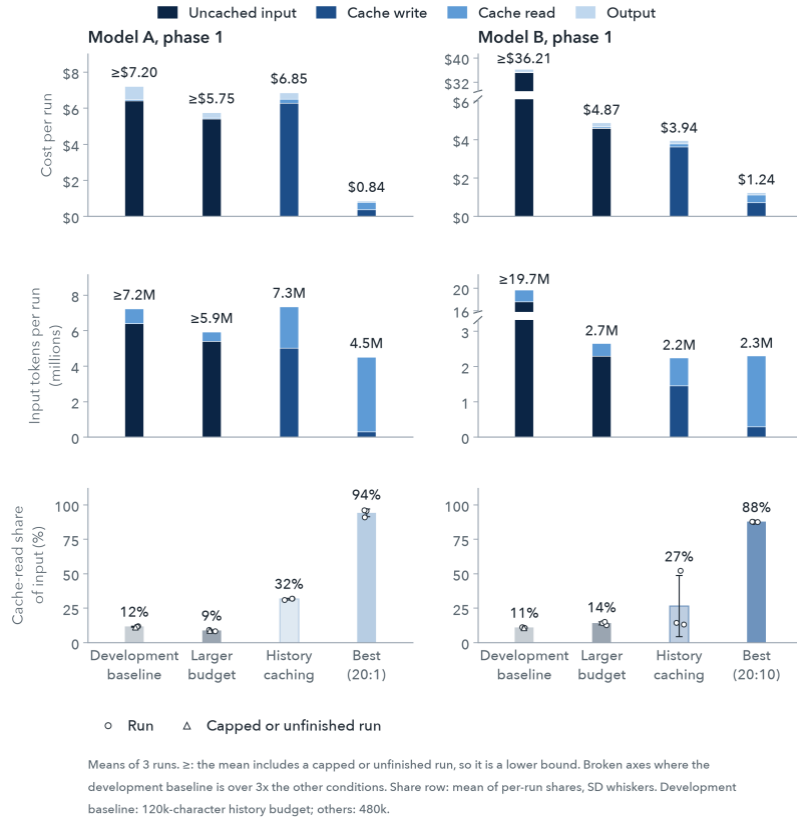


Figure S7. Cost breakdown for the four main conditions: cost and input volume by token type, and cache-read share as the mean of per-run shares, for the first phase (top) and the second (bottom).

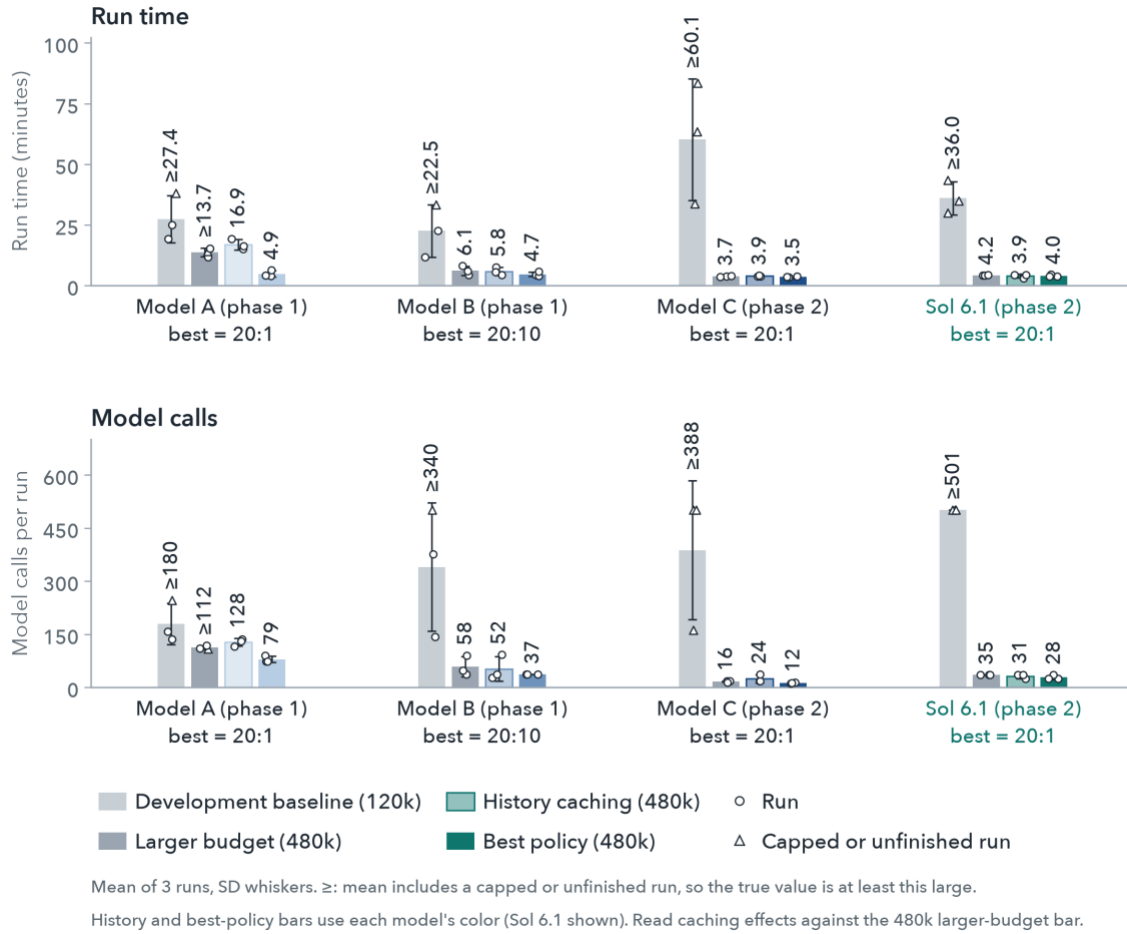


Figure S8. Run time and model calls per run for the four main conditions on all four models.

Table S1. Run flow per model.

	Model A	Model B	Model C	Sol 6.1
Selected	36	36	36	36
In plotted means	35	36	36	36
Excluded	1	0	0	0
Step or budget limit	0	4	14	15
Error or timeout	2	0	4	0
Finished	33	32	18	21
Lower-bound cost	2	4	18	15

Table S2. All selected runs, 36 per model on four models. Each cell lists runs 1, 2 and 3; ≥ marks a lower-bound cost and * the excluded Model A run (Table S3).

Model	Condition	Cost (USD)	Calls	Minutes	Cache read (%)
Model A	Baseline, 120k	≥10.16, 5.15, 6.30	246, 136, 157	38.1, 19.1, 25.0	11, 12, 12
	History only, 120k	6.97, 6.76, 7.28	157, 151, 164	20.5, 19.8, 20.5	17, 18, 19
	3:1, 120k	4.49, 14.34, 1.67	145, 329, 82	13.1, 53.5, 5.1	46, 31, 61
	10:1, 120k	2.69, 2.64, 2.60	141, 126, 114	8.6, 8.6, 6.3	74, 69, 65
	20:10, 120k	3.48, ≥0.02*, 3.32	109, 6, 120	7.6, 1.0, 7.2	62, 84, 67

Model	Condition	Cost (USD)	Calls	Minutes	Cache read (%)
	20:1, 120k	2.35, 2.29, 1.71	114, 117, 79	7.3, 8.2, 6.0	75, 78, 73
	Baseline, 480k	5.36, ≥ 5.56 , 6.32	110, 109, 118	11.6, 14.1, 15.2	9, 9, 8
	History only, 480k	7.28, 5.82, 7.44	136, 116, 131	15.1, 16.3, 19.3	31, 32, 32
	3:1, 480k	2.44, 2.14, 3.05	92, 115, 106	11.7, 9.0, 12.5	68, 77, 67
	10:1, 480k	1.61, 2.30, 1.63	143, 152, 137	8.6, 10.6, 9.7	89, 88, 91
	20:10, 480k	1.26, 1.47, 0.53	109, 120, 36	6.5, 7.5, 3.6	93, 92, 88
	20:1, 480k	0.62, 1.28, 0.62	74, 90, 74	4.0, 6.4, 4.2	96, 91, 96
Model B	Baseline, 120k	14.46, ≥ 53.99 , 40.18	143, 501, 376	11.7, 33.3, 22.6	11, 11, 11
	History only, 120k	27.67, 4.94, 23.76	213, 87, 184	17.6, 5.8, 13.3	11, 52, 12
	3:1, 120k	38.32, ≥ 60.58 , 7.80	316, 501, 174	25.5, 34.4, 12.6	22, 23, 71
	10:1, 120k	21.20, ≥ 46.04 , 24.99	174, 501, 204	16.4, 29.3, 17.4	28, 51, 29
	20:10, 120k	26.96, 20.08, ≥ 72.48	189, 141, 445	22.9, 11.3, 31.7	34, 31, 25
	20:1, 120k	7.67, 71.77, 55.37	72, 481, 380	7.5, 34.9, 27.1	40, 24, 26
	Baseline, 480k	2.98, 8.04, 3.60	37, 90, 48	4.3, 8.1, 5.9	14, 13, 15
	History only, 480k	5.20, 2.96, 3.66	92, 27, 37	7.6, 4.4, 5.6	52, 13, 14
	3:1, 480k	1.98, 1.55, 1.78	41, 42, 35	4.7, 4.4, 5.5	67, 76, 65
	10:1, 480k	2.42, 0.87, 1.00	70, 26, 27	8.5, 4.2, 4.7	87, 84, 82
	20:10, 480k	1.24, 1.25, 1.23	37, 37, 37	4.3, 4.0, 5.7	88, 88, 88
	20:1, 480k	2.38, 1.02, 0.95	67, 37, 37	7.7, 4.6, 5.0	89, 91, 92
Model C	Baseline, 120k	≥ 56.79 , ≥ 52.62 , ≥ 17.70	501, 501, 161	63.4, 83.4, 33.6	10, 12, 11
	History only, 120k	≥ 63.97 , ≥ 68.17 , ≥ 64.18	501, 501, 501	94.2, 82.8, 74.0	13, 13, 13
	3:1, 120k	≥ 69.92 , ≥ 72.34 , ≥ 72.46	501, 495, 497	91.2, 79.4, 75.6	16, 13, 13
	10:1, 120k	≥ 66.90 , ≥ 66.54 , ≥ 71.78	501, 501, 501	66.5, 93.6, 88.2	26, 26, 20
	20:10, 120k	≥ 72.34 , ≥ 38.75 , ≥ 72.40	472, 259, 484	86.3, 52.4, 114.9	29, 30, 30
	20:1, 120k	≥ 50.16 , ≥ 72.48 , ≥ 26.84	346, 458, 196	61.5, 79.2, 43.7	29, 22, 32
	Baseline, 480k	1.44, 1.48, 1.25	17, 17, 14	3.8, 3.9, 3.5	14, 14, 14
	History only, 480k	1.66, 1.77, 3.94	17, 18, 37	3.8, 3.8, 4.2	14, 15, 15
	3:1, 480k	1.52, 1.61, 1.39	15, 14, 13	3.9, 3.9, 3.6	20, 14, 15
	10:1, 480k	1.18, 0.82, 0.92	15, 13, 12	3.8, 3.8, 3.4	50, 65, 56
	20:10, 480k	0.70, 1.06, 0.98	12, 11, 10	2.8, 3.3, 3.2	72, 42, 37
	20:1, 480k	0.67, 0.67, 0.64	11, 14, 12	3.7, 3.4, 3.5	71, 76, 73
Sol 6.1	Baseline, 120k	≥ 35.59 , ≥ 34.88 , ≥ 35.85	501, 501, 501	29.8, 34.9, 43.3	9, 9, 9
	History only, 120k	≥ 44.13 , ≥ 44.43 , 43.10	501, 501, 484	35.5, 32.2, 35.9	9, 9, 9
	3:1, 120k	≥ 39.24 , ≥ 41.74 , 20.64	501, 501, 256	35.8, 34.4, 18.4	23, 18, 20
	10:1, 120k	≥ 45.44 , 33.68, ≥ 43.97	501, 377, 501	32.1, 25.7, 41.8	22, 23, 24
	20:10, 120k	≥ 59.75 , ≥ 59.21 , ≥ 62.01	501, 501, 501	38.9, 46.7, 38.2	25, 26, 20
	20:1, 120k	≥ 54.21 , ≥ 54.35 , ≥ 58.52	501, 501, 501	34.5, 35.0, 48.6	21, 21, 26
	Baseline, 480k	1.96, 1.97, 1.99	35, 35, 35	4.1, 4.2, 4.3	11, 11, 11
	History only, 480k	1.63, 2.53, 2.54	24, 35, 35	3.0, 4.4, 4.4	11, 10, 11
	3:1, 480k	1.11, 1.12, 1.11	35, 35, 35	3.8, 4.4, 4.1	64, 64, 64
	10:1, 480k	0.48, 0.61, 0.63	28, 35, 35	4.0, 4.4, 4.4	85, 85, 85
	20:10, 480k	0.73, 0.71, 0.71	36, 35, 35	4.2, 4.5, 4.3	88, 88, 88
	20:1, 480k	0.45, 0.51, 0.46	25, 35, 24	3.7, 4.5, 3.8	88, 91, 88

Table S3. Results with and without the excluded Model A run. This is not one of the four main-text conditions.

Model A 20:10, 120k	With the excluded run, 3 runs	Without it, 2 runs
Mean cost per run	\$2.27	\$3.40
Mean model calls	78	114
Mean minutes	5.3	7.4

Table S4. Mean cost per run by condition for all four models. $\geq$ marks lower bounds from capped or unfinished runs; * marks a mean without the excluded Model A run (Table S3).

Condition	Model A	Model B	Model C	Sol 6.1
Baseline, 120k	$\geq \$7.20$	$\geq \$36.21$	$\geq \$42.37$	$\geq \$35.44$

Condition	Model A	Model B	Model C	Sol 6.1
History only, 120k	\$7.00	\$18.79	≥\$65.44	≥\$43.89
3:1, 120k	\$6.83	≥\$35.57	≥\$71.57	≥\$33.87
10:1, 120k	\$2.64	≥\$30.74	≥\$68.41	≥\$41.03
20:10, 120k	\$3.40*	≥\$39.84	≥\$61.16	≥\$60.32
20:1, 120k	\$2.11	\$44.94	≥\$49.83	≥\$55.69
Baseline, 480k	≥\$5.75	\$4.87	\$1.39	\$1.97
History only, 480k	\$6.85	\$3.94	\$2.46	\$2.23
3:1, 480k	\$2.54	\$1.77	\$1.51	\$1.11
10:1, 480k	\$1.85	\$1.43	\$0.97	\$0.57
20:10, 480k	\$1.09	\$1.24	\$0.91	\$0.72
20:1, 480k	\$0.84	\$1.45	\$0.66	\$0.47

Table S5. Provenance.

Item	Model A	Model B	Model C	Sol 6.1
Phase	1	1	2	2
Execution window of analyzed runs	21 Sep 16:42 to 22 Sep 02:16 UTC	22 Sep 13:05 to 23 Sep 05:22 UTC	30 Sep 03:57 to 30 Sep 12:57 UTC	30 Sep 06:08 to 30 Sep 13:58 UTC
Runs executed / analyzed	36 / 35	36 / 36	36 / 36	36 / 36
Receipted provider requests (analyzed runs)	4,455	5,906	8,149	9,224
Requested model ID	model-a	model-b	model-c	gpt-6.1-sol
Returned model ID	model-a	model-b (per-call wire model)	model-c	gpt-6.1-sol
Provider API	Other provider's messages API, direct	Other provider's messages API, direct	Other provider's messages API, direct	OpenAI Responses
Route	StackAI workflow, BrowserNav action, E2B sandbox	Same	Regular workflow inference API, same action and sandbox	Same as Model C
Backend source commit	f1b0e5c	c5aead0 with wheel b7d01d1 (19 runs); 69103a1 with wheel 23abb99 (17 runs)	fb9275f at start, 9ea6ef9 after disclosed persistence fixes (base c5aead0)	Same as Model C
BrowserNav wheel SHA-256	4b01f52	See source row	a6fc26b (one wheel, whole phase)	Same as Model C
Sandbox template	bcdjkhz4x1vqpcvbi3fe	bcdjkhz4x1vqpcvbi3fe	bcdjkhz4x1vqpcvbi3fe	bcdjkhz4x1vqpcvbi3fe
Reasoning effort	Not set (omitted)	Not set (omitted)	high (request effort field)	high (reasoning.effort)
Thinking	Not set; model default	Not set; model default (thinking tokens on 17% of requests)	adaptive thinking; signed thinking replayed for retained turns	Encrypted reasoning replayed for retained turns (include=reasoning.encrypted_content)
Service tier requested / returned	Not in recorded settings / not recorded	Not in recorded settings / not recorded	standard_only / not returned (8,149)	default / default (9,224)
Max output tokens; max steps	16,384; 500	16,384; 500	16,384; 500	16,384; 500
Command timeout; provider timeout	14,400 s; SDK 120 s, SDK retries off	14,400 s; 600 s	14,400 s; 600 s	14,400 s; 600 s
Model-cost cap per run	\$20	\$75	\$75	\$75
Configured input admission limit (tokens)	200,000	1,000,000	1,000,000	922,000
Price table version	provider-model-a-2026-09-21	provider-model-b-2026-09-21	provider-model-c-2026-09-29	openai-sol-6-1-standard-2026-09-29
Cache mode by policy	Baseline: current (marker on system prompt); history policies: history (adds marker on newest tool result)	Same	Same	prompt_cache_options.mode=explicit; breakpoints on the same two points
Cache lifetime	5 min (300 s)	5 min (300 s)	5 min (300 s)	30 min (1,800 s)
Cache protocol	parallel_shared	parallel_shared	parallel_shared	parallel_shared
Tool choice for read and done	Forced on some transitions, parallel tool use off	Forced on some transitions, parallel tool use off	Automatic with the same instruction, parallel tool use off (forcing was incompatible)	Native forced function choice, parallel tool calls off; strict=false
Other notable settings	None beyond the policy fields	None beyond the policy fields	thinking prefix mismatch handled by dropping the block (provider beta flag);	store=false; truncation=disabled; image detail high; one earlier checkpoint kept while its prefix

Item	Model A	Model B	Model C	Sol 6.1
			partial text answer kept	matches

Table S6. Input tokens per request by model for the four main conditions, including images, the system prompt and tools.

Model	Condition	Requests	Median	p90	Max
Model A	Baseline, 120k	539	45,133	49,068	51,519
	Baseline, 480k	337	47,508	99,001	119,507
	History only, 480k	383	52,255	108,924	141,429
	20:1, 480k	238	55,886	90,565	115,130
Model B	Baseline, 120k	1,020	60,076	61,044	61,879
	Baseline, 480k	175	47,992	67,232	81,138
	History only, 480k	156	43,682	64,871	86,015
	20:10, 480k	111	69,983	100,773	109,880
Model C	Baseline, 120k	1,163	58,195	61,352	63,310
	Baseline, 480k	48	40,942	83,557	91,672
	History only, 480k	72	43,236	77,480	85,978
	20:1, 480k	37	54,201	107,206	116,753
Sol 6.1	Baseline, 120k	1,503	39,028	39,777	42,811
	Baseline, 480k	105	30,206	51,146	55,350
	History only, 480k	94	31,214	50,955	54,798
	20:1, 480k	84	47,144	69,917	79,031

Table S7. Follow-up without screenshot pruning at 480,000 characters, by run.

Model	Run	Calls	Cost (USD)	Minutes	Cache read (%)	Saw 192	Correct total
Model B	1	39	1.07	5.1	95	176 of 192	yes
	2	48	1.30	5.1	93	yes	yes
	3	25	0.74	4.5	92	yes	yes
	4	36	0.97	4.8	95	yes	yes
Model C	1	11	0.60	4.1	80	yes	yes
	2	14	0.67	4.1	86	yes	yes
	3	13	0.62	3.6	85	yes	yes
	4	11	0.61	3.8	81	yes	yes
Sol 6.1	1	28	0.40	4.7	93	yes	yes
	2	35	0.49	5.1	95	yes	yes
	3	35	0.49	4.9	95	yes	yes
	4	35	0.49	4.8	95	yes	yes

6.1 Reproducibility details

Runs were orchestrated on a MacBook Pro with an Apple M5 Max processor, 18 CPU cores and 64 GB of memory, with no inference running locally. The final cohorts used one StackAI frontend and backend with shared Redis and the remote development database; earlier phases used separate instances in isolated Git worktrees.

In the first phase, archived requests and the installed runtime code omit thinking, effort and sampling settings. The model normally chose its tools, but the agent forced its read or done tool on some transitions, disabling parallel tool use on those calls. Runs were capped at 500 steps and 501 calls, with a 14,400-second command timeout, a 600-second provider timeout, 16,384 output tokens and a five-minute cache lifetime. An exact reproduction also requires each run's source

revision, package checksum, sandbox template, workflow graph, prompt and effective settings, all recorded in the per-run manifests.

6.2 Earlier experiments

- **Tool-history caching in the browser.** Single runs on three browser tasks with Model A, showing both reuse in principle and the effect of screenshot edits.
- **Earlier stable checkpoint.** A marker placed before the changing screenshot content. Some calls reused more cached input, but whole-run cost rose in the live pairs.
- **Retention-only pilot.** More screenshots retained without extending the cache boundary. Cache reads stayed flat and uncached input doubled, so batch pruning is always paired with history caching here.
- **Replays.** Fixed-request replays measured cost separately from agent behavior, because the model's answers did not feed back into the next request. They are not pooled with this paper's results.