Published Writings

- AI Didn't Remove the Hard Part of Coding. It Moved It.
- METR 2025 RCT: AI Tools Slowed Experienced Open-Source Devs by 19% — Analysis
- We Spent $18 Benchmarking Self-Hosted LLMs. The Cheapest APIs Still Won — Except When They Didn't.
- Rent the Machine or Buy the Tokens? We Spent $18 Finding Out.
- Stop Comparing GPU Providers. Count Your Tokens.
- The Code Writes Itself. The Judgment Doesn't.
- The Models Got Smarter. We Didn't Change How We Talk to Them.

AI Didn't Remove the Hard Part of Coding. It Moved It.

For decades, the bottleneck in software was producing the answer. That bottleneck is gone, and most developers are still training for it.

The skill we all optimized for

Everything about how developers learn was built around *know-how*: memorize the syntax, learn the patterns, practice until you can produce working code from a blank file. Interviews test it. Courses teach it. Careers were ranked by it.

Then LLMs made production nearly free. Describe what you want, and working-looking code appears in seconds. By Google's own account, more than a quarter of its new code is now AI-generated. The blank file is no longer the enemy.

It's tempting to conclude the job just got easier. The real question changed underneath us: if anyone can *generate* an answer, what separates a good developer from a bad one?

The scarce skill is no longer producing the answer. It is seeing, quickly and reliably, whether an answer is any good, and knowing how to critique the model into a better one. Call it *know-see*. Know-how ships the first draft. Know-see decides whether that draft belongs in production.

Why we keep training the wrong muscle

Because production was expensive for fifty years, our instincts equate typing with working and output with progress. Review was the low-status activity, the thing you did to other people's code, quickly, before getting back to "real work." Nobody built a career on being great at reading code. That habit is now exactly backwards.

What the evidence says

Generation is cheap. Verification is the new cost center. The METR randomized trial followed experienced open-source developers on their own mature codebases and found something uncomfortable: with AI tools they took 19% longer to finish tasks, while believing they had been 20% faster. Where did the time go? Not into typing. Developers accepted fewer than 44% of AI suggestions, and a majority reported major cleanup on the code they did accept. Reading, judging, and fixing, the know-see work, quietly absorbed the savings.

Your feeling of productivity is not a measurement. That same study exposed a roughly 40-point gap between perceived and actual speed. Google's DORA report, drawing on 39,000 professionals, found the same pattern at scale: as AI adoption rose 25%, delivery speed dipped and system stability dropped 7.2%, while three-quarters of developers *felt* more productive. Know-see starts with distrusting the feeling. Instrument reality: task completion times, defect rates, review depth. If you can't see your own performance clearly, you certainly can't see the model's.

The best output goes to the best critics. An LLM is like an infinitely fast junior colleague with unlimited confidence: the first draft arrives in seconds, polished-looking, occasionally wrong in ways designed to be missed. What improves the next draft isn't a vaguer "make it better", it's a precise critique. Name the flaw: "this ignores the timezone edge case," "this duplicates the retry logic in the client," "this test asserts nothing." The developers who get remarkable output are running a tight loop: define what *good* looks like before prompting, review the result like a hostile senior reviewing a junior's PR, and feed back specific, named defects. Being the main critic is not overhead on the AI workflow. It *is* the workflow.

Two fair objections. "*Other studies show real gains.*" True, a large study across Microsoft and Accenture found developers with Copilot completed 26% more tasks. Both findings can hold: gains concentrate where tasks are self-contained and verification is cheap; losses concentrate in complex, interconnected systems where seeing a subtle flaw is hard. The variable separating the two studies is precisely the cost of judgment. "*Models will get good enough that checking won't matter.*" The better models get, the more their failures shift from obvious to plausible, wrong in ways that read as right. Improvement raises the bar for the critic; it doesn't retire the critic.

A quick self-test

1. Can you tell within a minute whether a generated function is production-worthy, or do you find out in review, or worse, in production?
2. Do you read AI output with the same rigor you'd apply to a junior's pull request, or with the leniency you'd apply to your own code?
3. When output is mediocre, do you re-prompt with a named, specific critique, or accept it and patch by hand?

4. Do you measure your AI-assisted work, or trust how fast it feels?

Training know-see, deliberately

Read more code than you write. Reviewing, hostile, line-by-line, "what would break this?", is now the core practice, not the chore.

Write the definition of done before you prompt. If you can't state what good looks like, you'll accept whatever looks finished.

Build a critique vocabulary. Models respond to precision. "This is wrong" produces a reshuffle; "this leaks the connection on the error path" produces a fix.

AI didn't make developers replaceable. It made their judgment the entire product.

Sources: METR, "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity" (RCT, 2025); Google DORA Report 2024; MIT/Princeton/UPenn Copilot field study; Alphabet Q3 2024 earnings call.

METR 2025 RCT: AI Tools Slowed Experienced Open-Source Devs by 19% — Analysis

> Saved from a study session on 2026-07-08. Source: arXiv 2507.09089v2 — "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity" (Becker, Rush, Barnes & Rein, METR, July 2025).
>
> PDF: Clippings/2507.09089v2.pdf

What the study is

METR ran a **randomized controlled trial** — the first real RCT of AI coding tools on *real* work — between February and June 2025:

- **16 experienced open-source developers**, working on repos they'd contributed to for ~5 years (avg. 1,500 commits). These are big, mature projects: ~23k stars, ~1.1M lines of code, strict review standards.
- They brought **246 real issues** from their own repos — bugs, features, ~2 hours of work each.
- **Each issue was randomized by coin flip**: AI-allowed (mostly Cursor Pro + Claude 3.5/3.7 Sonnet) or AI-disallowed. Devs recorded their screens and reported time spent.

The clever part of the design is *when* things get measured. Before randomization, developers forecast how long each issue would take both with and without AI. That does two jobs: it gives a difficulty measure that's independent of the coin flip, and it captures expectations. The outcome — time to complete the issue — is **fixed before treatment assignment**, so AI can't game the metric. Compare that to earlier studies which measured lines of code or number of PRs — things AI can inflate without any real productivity gain (more verbose code, work split into more PRs).

The headline result

Three numbers, and the gap between them is the whole story:

Who	Expected effect of AI
Economics experts	39% faster
ML experts	38% faster
The developers themselves (before)	24% faster
The developers themselves (*after doing the work*)	20% faster
What actually happened	**19% slower**

Everyone got the *sign* wrong. And the most unsettling row is the fourth one: after spending hours using the AI on real tasks, developers still believed it had sped them up by 20% — while it had actually slowed them down. Interestingly, the devs' forecasts of *task duration* were well-calibrated (correlation ~0.6 with actual times) — they know how long work takes; what's inverted is specifically their belief about AI's contribution.

Where the time went

They hand-labeled 143 hours of screen recordings at ~10-second resolution. With AI allowed, developers spent *less* time actively coding and searching, and instead spent time **prompting the AI, waiting on generations, reviewing AI output** — and notably more time *idle*. The AI didn't remove work; it swapped coding time for AI-babysitting time, and the swap was net negative in this setting.

Why? The factor analysis

They examined 21 possible explanations, grouped into four types (direct productivity loss / experimental artifact / raises human performance / limits AI performance). Five factors had real evidence **for** contributing to slowdown:

1. **Over-optimism** — devs believed AI was helping, so they kept using it past the point of usefulness.
2. **High familiarity** — these devs are experts *in these exact repos*; it's hard for AI to beat someone with all the context already in their head. Slowdown was worse on tasks devs knew best.

3. **Large, complex repos** — AI performs worse in million-line codebases than in greenfield projects.
4. **Low AI reliability** — devs accepted **<44%** of generations; 75% read every line of AI output; 56% often made major cleanups. ~9% of AI-allowed time went just to reviewing/cleaning AI output.
5. **Missing tacit context** — "AI acts like a new contributor": it doesn't know the undocumented constraints, the weird backwards-compat case, which location is the *right* place for an edit.

Six factors had evidence *against* (cheating, dropout, non-frontier models, unfamiliar IDE, etc.), and ten were unclear. The authors are careful: they can't fully rule out experimental artifacts, but the slowdown was robust across many alternative analyses.

The caveat they insist on

This does **not** say "AI doesn't speed up developers." It says: for *experts* working in *codebases they know deeply* with *high quality bars*, early-2025 AI slowed them down. The same paper explicitly says results are consistent with big speedups on greenfield projects or unfamiliar code — and their appendix quotes back this up: devs found AI *most* helpful precisely on tasks they'd never done before ("first time with Git hooks, AI saved me 3 hours").

Open discussion threads

1. **The perception gap** — developers finished the study still believing they'd been sped up 20%. How can a tool slow you down while feeling helpful? (Candidate hypothesis: "trading speed for ease" — effort vs. time.)
2. **Self-mapping** — for a heavy AI-tool user: which side of the familiarity line are you usually on, and do you recognize the review-and-cleanup tax in your own sessions?
3. **Design critique** — 16 developers, issues capped at ~2 hours: what does that exclude?

We Spent $18 Benchmarking Self-Hosted LLMs. The Cheapest APIs Still Won — Except When They Didn't.

The same GPU-hour, the same model, the same task: 122,733 requests one afternoon, 56,987 the next. Nothing about the infrastructure changed. Only the traffic did.

The question everyone prices wrong

Every team building on open models eventually faces the same fork: pay a provider per token, or rent a GPU by the hour and serve the model yourself. The debate usually runs on the wrong variables — model quality, platform features, vendor preference. So we stopped debating and measured it.

Our setup was deliberately simple. Modal's new Auto Endpoints deploy an open model behind an OpenAI-compatible API in one command, billed per second of GPU uptime, scaled to zero when idle. We deployed two models at opposite ends of the scale — Gemma 4 E2B (a 2B-class model on one A100, ~$2.50/hour) and OpenAI's GPT-OSS-120B (117B parameters, on one B200 at $9/hour) — and ran each through an identical one-hour protocol: five minutes at concurrency 1 to measure what a single user feels, fifteen at concurrency 16 to simulate a realistic pipeline, forty at concurrency 48 to find the ceiling. The task was sentiment classification with strict JSON-schema output — intentionally easy, because we were testing the infrastructure and its economics, not the model's intelligence. One variable at a time.

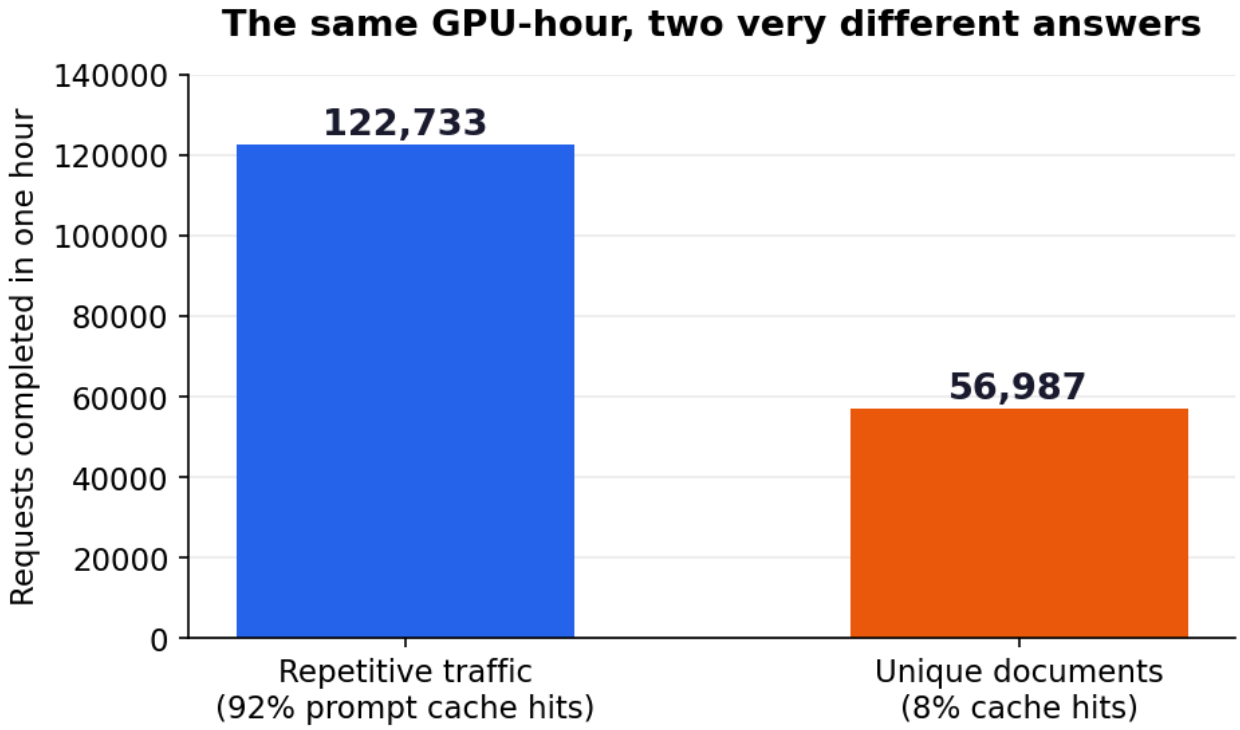
Five experiments. Roughly 345,000 requests. Total spend: about $18.

What we found: whether self-hosting beats the API is not a model question or a platform question. It is a traffic question — how repetitive your prompts are, how heavy

your volume is, and how steady it runs.

The 2.2x nobody puts on a pricing page

Our first run cycled a fixed set of 48 texts for an hour and completed 122,733 requests. Then we reran the identical hour with one change: every request carried a unique document. Throughput fell to 56,987 — a 2.2x drop, on the same GPU, the same model, the same configuration.



Gemma 4 E2B-it, 1×A100-40GB ($2.50/hr), identical configuration — only the traffic changed. 2.2× difference.

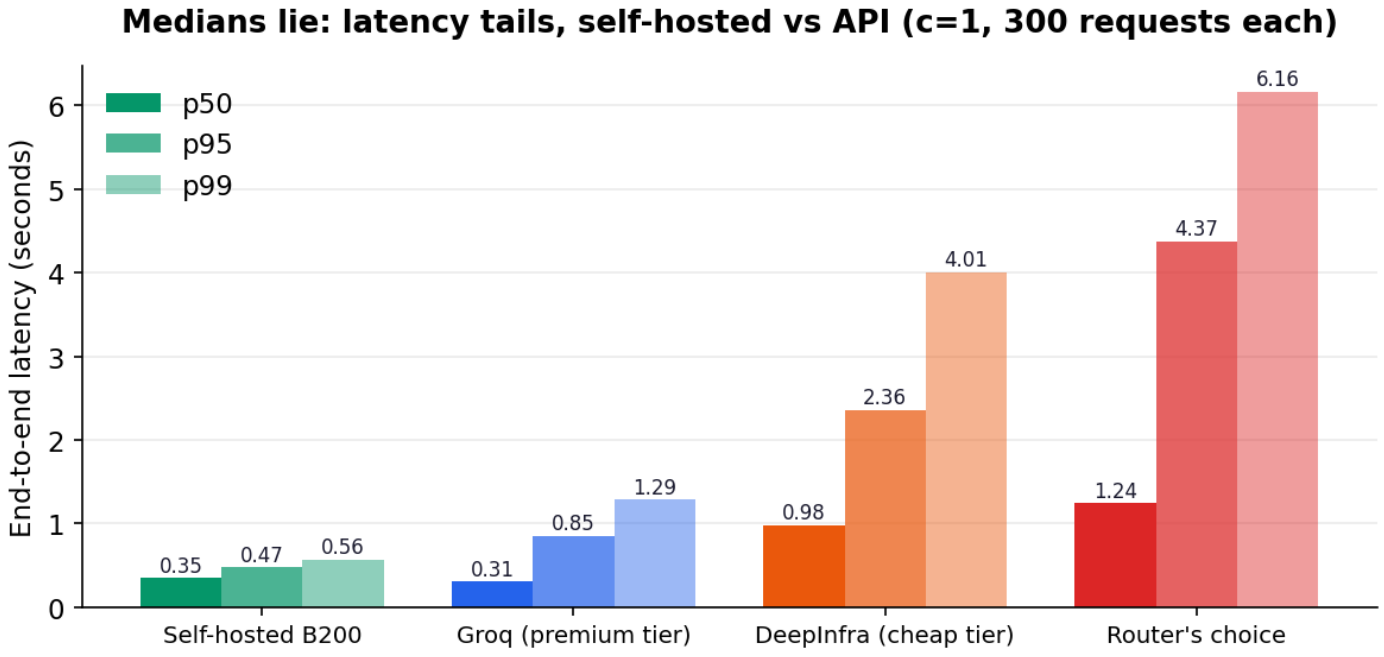
The mechanism is prefix caching. Modern inference engines cache the computation for repeated prompt prefixes; the engine's own metrics showed a 92% cache hit rate in the first run and 8–11% in the second (just the shared system prompt). When your traffic repeats — shared instructions, templated extraction, RAG over a stable corpus — most of your "input tokens" cost the GPU almost nothing.

Here is why that matters commercially: per-token APIs charge you the same for a cached token and a computed one. When you rent the GPU, the caching dividend is yours; when you pay per token, it is your provider's margin. A team whose traffic is 90% repetitive and a team processing unique documents are buying completely different quantities of compute — and only one pricing model lets them see it.

The scale surprise

Intuition says a 50x bigger model costs vastly more to run. Our invoices disagree. The 120B model on the B200 delivered 3.7x the throughput of the 2B on the A100 — 33,000 versus 8,900 tokens per second on unique documents — at 3.6x the hourly price. Cost per million tokens came out flat: $0.076 versus $0.078. Cost per thousand requests fell threefold, to about four and a half cents.

Two factors absorbed the entire scale jump: the model's mixture-of-experts design activates only 5.1B of its 117B parameters per token, and the newer GPU generation brings far more memory bandwidth per dollar. We state this as a measured result for this pairing, not a law — a dense 70B model would tell a different story. But it breaks a reflex worth breaking: do not price a model by its parameter count.



GPT-OSS-120B, same client, same payload, July 2026. Bars per target: p50 / p95 / p99.

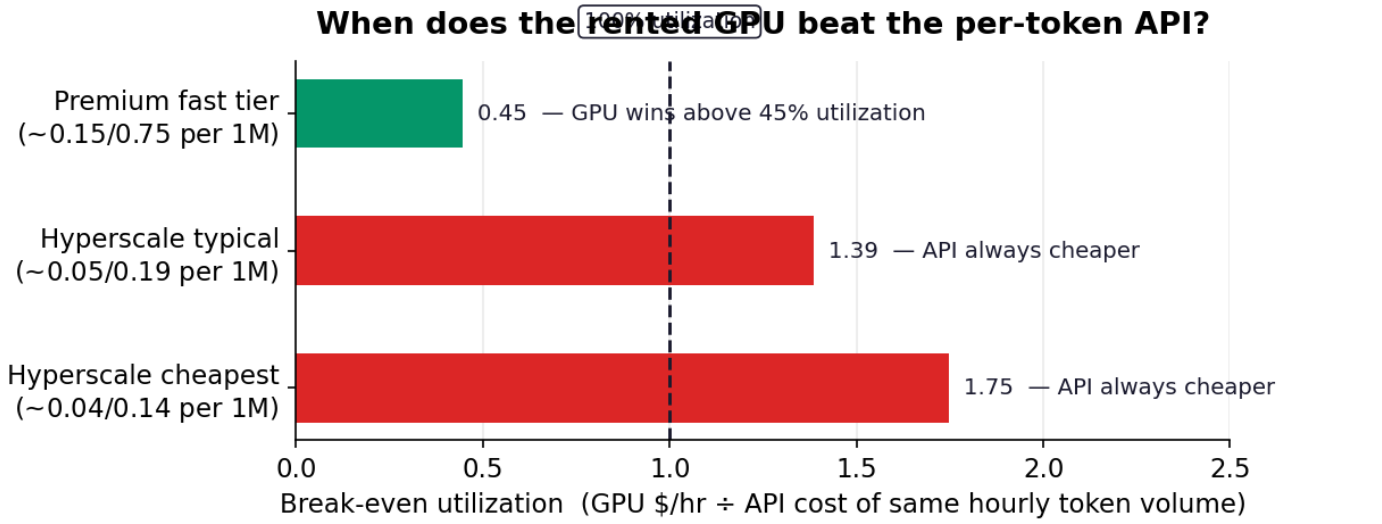
Where we lost, and where the equation flips

Now the result that a vendor-sponsored benchmark would bury. At our best measured throughput, the effective cost of the self-hosted 120B was $0.076 per million tokens. The cheapest hyperscale API listings for the very same model sit near $0.04 in and $0.14–0.18 out — meaning that for unique-document traffic, those providers beat our single GPU even if we kept it saturated 100% of the time. Break-even never arrives. The reason is structural: a provider batching requests from thousands of customers achieves a utilization no single tenant can, and one GPU cannot out-pool a

fleet.

But "the API" is not one price. Listings for this same model spread across roughly an 8x range between hyperscale batchers and premium low-latency tiers. Against the fast tier ($0.15 in / $0.75 out per million), our B200 broke even at 31–45% utilization and won decisively beyond it. And on repetitive traffic — the caching regime — the self-hosted option beat most of the market outright.

So the honest map has three regions. Unique documents at moderate volume: buy tokens from a hyperscale provider. High-volume repetitive traffic, or workloads currently priced at premium API tiers: the hourly GPU wins. Everything else: it depends on one number you can compute in advance — **break-even utilization**: your GPU's hourly price divided by what your hourly token volume would cost at the API. Ours ranged from 0.31 (self-hosting wins easily) to 1.86 (it cannot win).



GPT-OSS-120B on 1×B200 at $9/hr, measured saturation throughput (~119M tokens/hr, 98% input). July 2026 list prices.

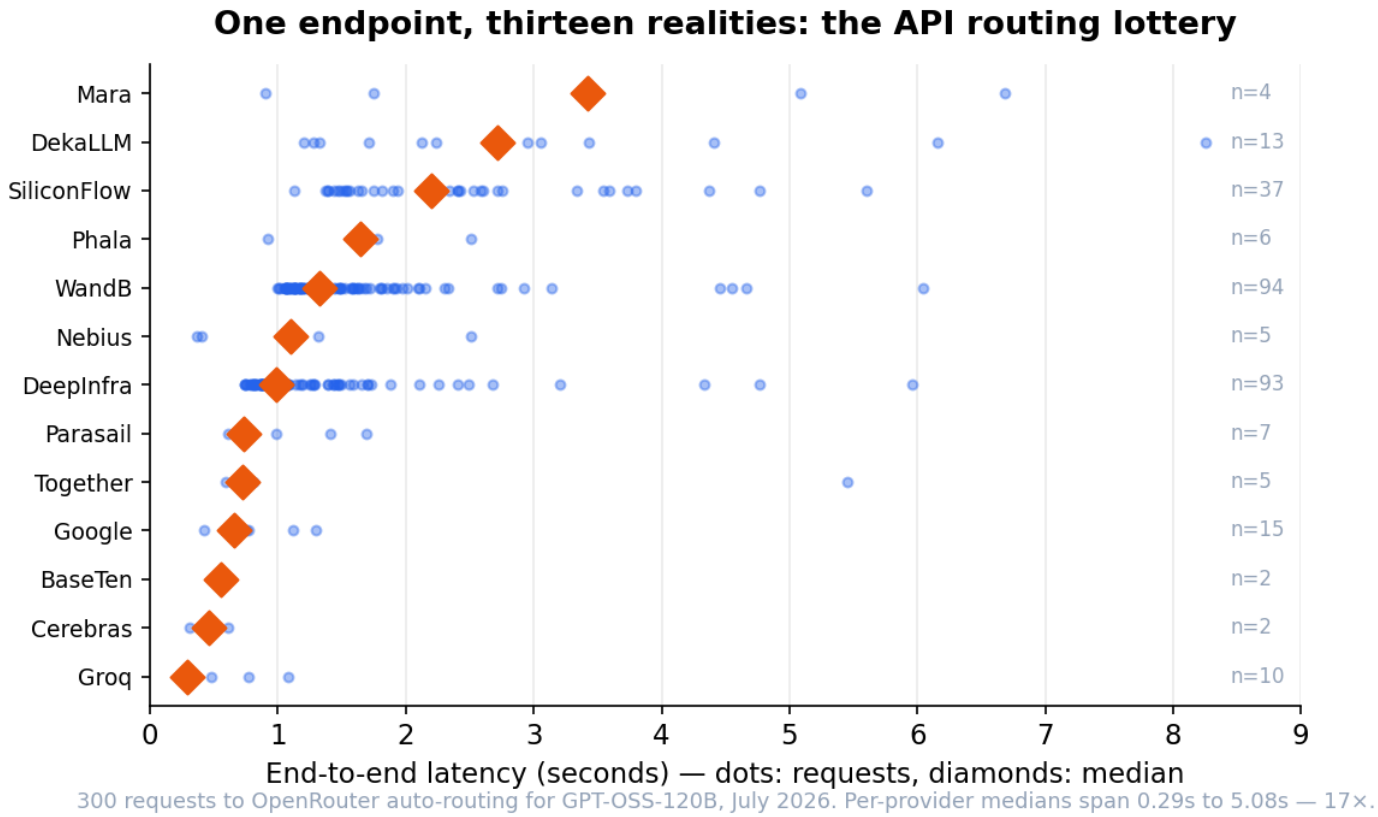
One scope note: our workload was 98% input tokens — classification over documents. Output-heavy workloads like chat or drafting shift the math on both sides, which is exactly why you should run the calculation on your own traffic mix. The prices in this article will age; that division will not.

How fast can one GPU go

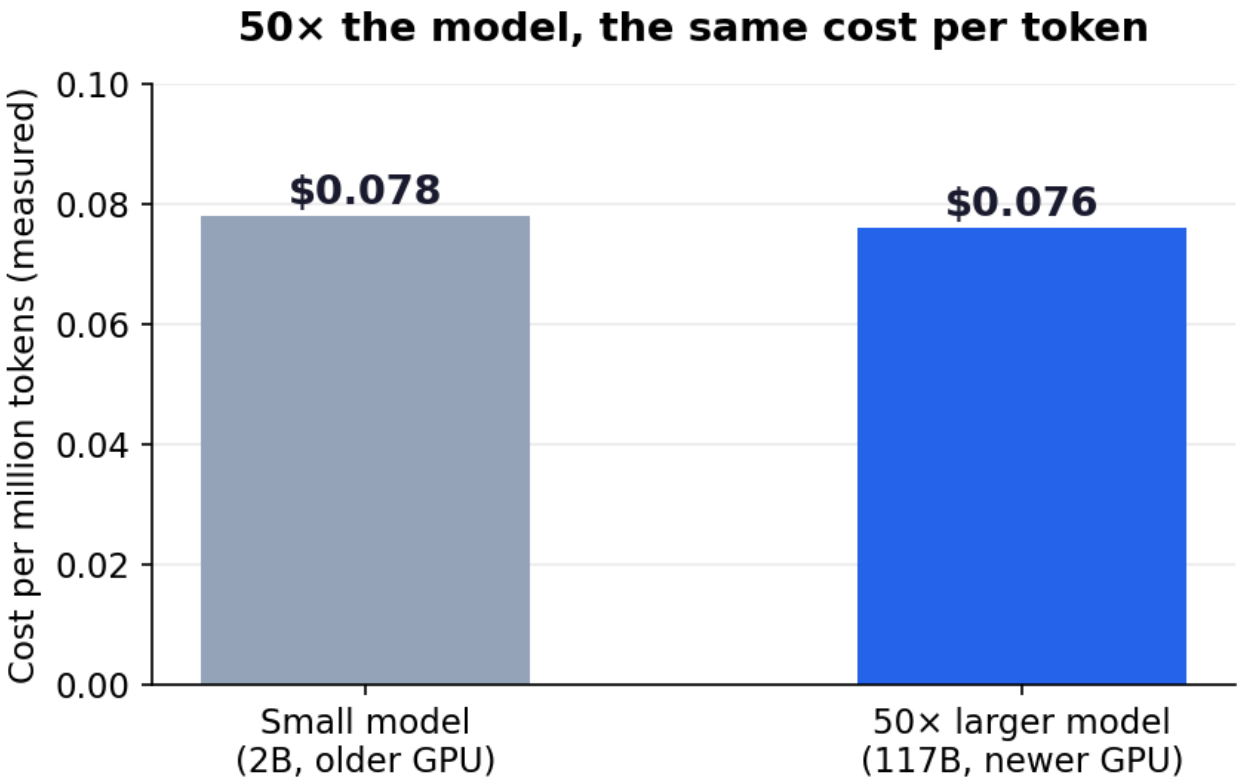
Cost is only half the fork; the other half is speed. So we ran a final side-by-side: 300 identical requests at concurrency 1, from the same machine, against our self-hosted B200 and against the same model on the API market via OpenRouter — once letting the router choose the provider, once pinned to the cheapest tier (DeepInfra), once pinned to the fastest (Groq).

End-to-end, c=1	p50	p95	p99	Failures
Self-hosted B200	0.35s	0.47s	0.57s	0
Groq (premium tier)	0.31s	0.85s	1.29s	2 (rate limits)

End-to-end, c=1	p50	p95	p99	Failures
DeepInfra (cheap tier)	0.98s	2.36s	4.01s	0
Router's choice	1.24s	4.37s	6.16s	7



Three things in that table are worth a contract review. First, the premium tier beat our median by 44 milliseconds — and lost everywhere else: our p95 was almost twice as good, our p99 was 2.3x better, our jitter four times tighter, and we were never rate-limited. Dedicated tenancy does not buy you the fastest median; it buys you a tail that barely exists, and tail latency is what users actually feel. Second, "the API" as most users consume it — the default route — is a lottery: our requests were served by thirteen different providers whose individual medians spread from 0.29 to 5.08 seconds, a 17x range behind a single endpoint.



Measured at maximum sustained load, unique-document traffic, July 2026. Hardware rented per hour.

Third, an accidental finding we consider the most underreported risk in this market: every one of the seven failures on the default route was a malformed structured-output response, and all seven came from one provider — a 32% schema failure rate on the requests it served, against zero malformed responses in our 345,000 self-hosted requests. Structured-output reliability is a provider property, and nobody's pricing page mentions it.

Note the symmetry with the economics: the only API tier that matches a dedicated GPU's speed is the premium one — precisely the tier our break-even said self-hosting beats from 45% utilization. The more your product depends on latency, the less you are competing with the API market's floor, and the better the hourly GPU looks.

What the pricing page doesn't tell you

Three operational facts from our logs, because a field report owes you the friction. Cold starts ranged from four to nine minutes across four boots — a 2x variance — most of it engine initialization and CUDA-graph capture, not weight loading — which makes scale-to-zero unsuitable for interactive workloads without warm-up strategies. Under sustained overload with unique documents, the small model's serving recipe crashed and self-healed, costing eight minutes of availability mid-run; the 120B configuration processed 164,641 requests with a single error. And across all 344,000 requests, strict JSON-schema output never produced one malformed response —

structured output at the engine level is, in our experience, production-grade.

None of this appears in per-token pricing, and all of it appeared in our first three GPU-hours. That asymmetry is the deeper argument for running the test: the hourly model shows you your system's actual behavior — its caches, its ceilings, its failure modes — because you own the serving stack that produces it.

Four questions before your next inference decision

1. What fraction of your prompt tokens repeat across requests — and has anyone measured it?
2. What would one hour of your real token volume cost at your current provider's rates?
3. What is your break-even utilization against the cheapest and the fastest API tier serving your model?
4. Is your latency budget defined at the median or at the 99th percentile — and which tier of the API market actually meets it?

Our entire protocol — polling, phased load, token accounting, the break-even math — is a pair of Python scripts we have open-sourced [repo link]. Running it against your own endpoint costs about the price of lunch.

The API sells you tokens. The GPU sells you an hour. Which one is cheaper is written in your traffic, not on their pricing page.

Benchmarks: Modal Auto Endpoints, July 2026 — Gemma 4 E2B-it (1×A100-40GB, $2.50/hr) and GPT-OSS-120B (1×B200, $9/hr), SGLang recipes, us-west, one container, identical 3-phase protocol. API reference prices from public provider listings, July 2026. Latency comparison: 300 requests per target at concurrency 1 via OpenRouter (auto, DeepInfra-pinned, Groq-pinned), same client and payload; completion-token counts varied by provider (19-35 tokens) as reasoning controls are honored differently. Full per-request data and scripts: [repo].

Rent the Machine or Buy the Tokens? We Spent $18 Finding Out.

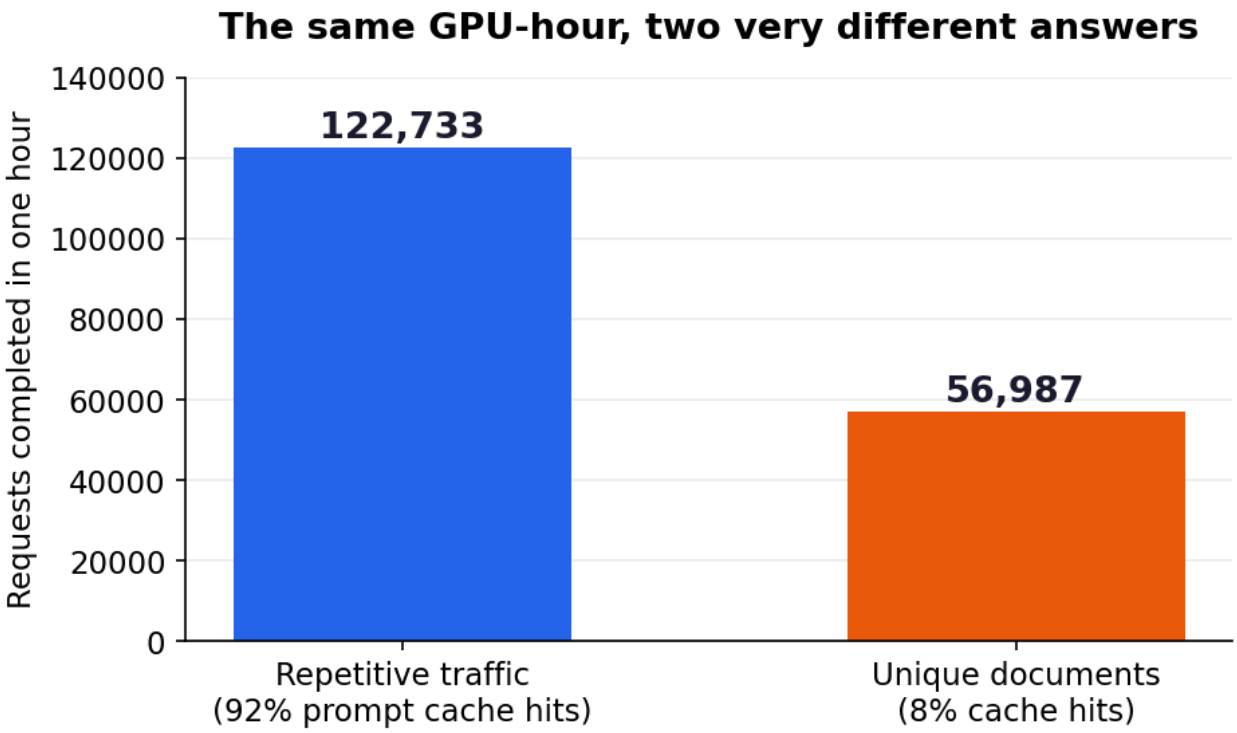
Every company building with AI eventually faces the same procurement fork: pay a provider for each unit of AI work (per-token pricing, like a taxi meter), or rent dedicated computing hardware by the hour and run an open model yourself (like leasing a car). The debate is usually settled by opinion. We settled it with invoices.

What we did

We took two freely available AI models — one small, one 50 times larger — and deployed each on rented GPU hardware in a single afternoon, using a platform that bills by the second and charges nothing when idle. We then pushed roughly 345,000 real requests through them under a controlled, repeatable test, and compared the resulting costs and speeds against the leading pay-per-token providers serving the exact same models. Total research budget: about $18.

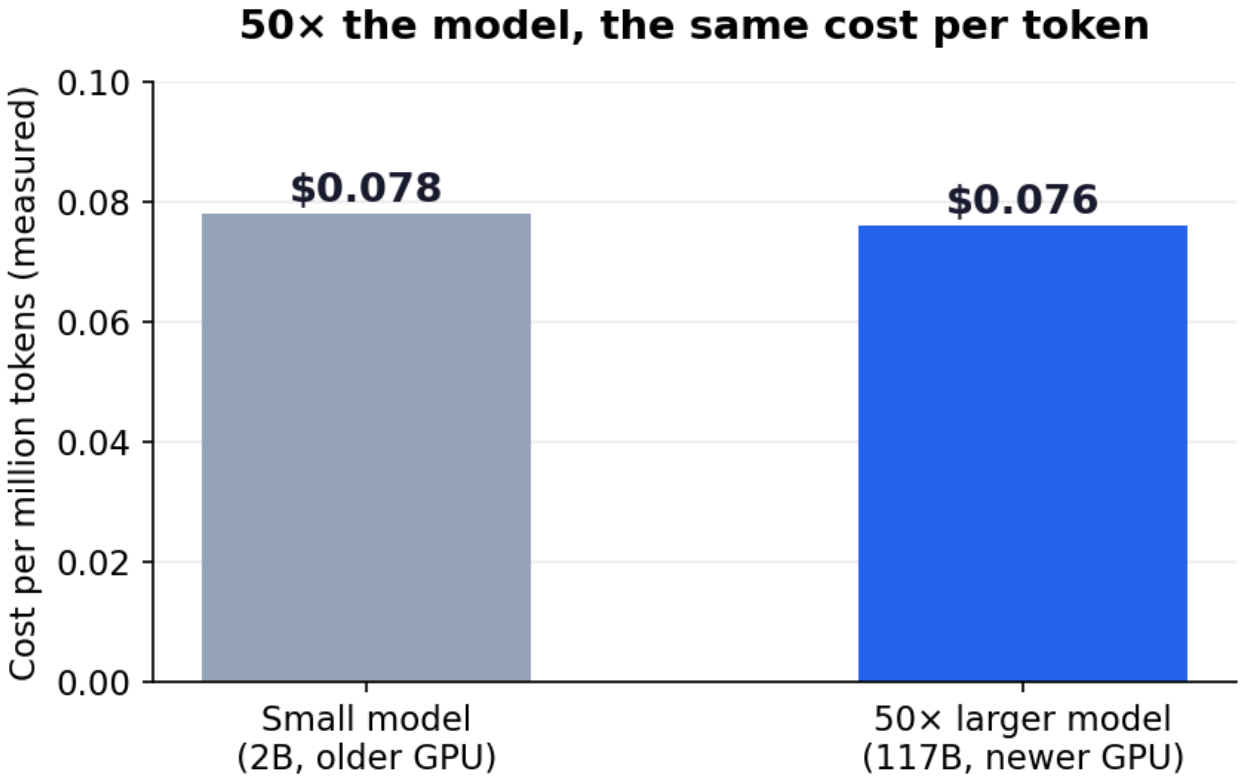
The three findings that matter

1. Your traffic pattern — not the model, not the vendor — decides which option is cheaper. The same hour of rented hardware processed 122,000 requests one day and 57,000 the next. The only difference: the first day's requests were repetitive, and modern AI infrastructure recognizes and reuses repeated work. Here is the commercial catch: per-token providers charge you full price for reused work. When you rent the machine, that efficiency is your savings; when you pay per token, it is your supplier's margin. Companies with repetitive AI workloads — templated document processing, standardized extraction, assistants with fixed instructions — are systematically overpaying under per-token pricing, and most have never measured by how much. In our test, the difference was 2.2x.



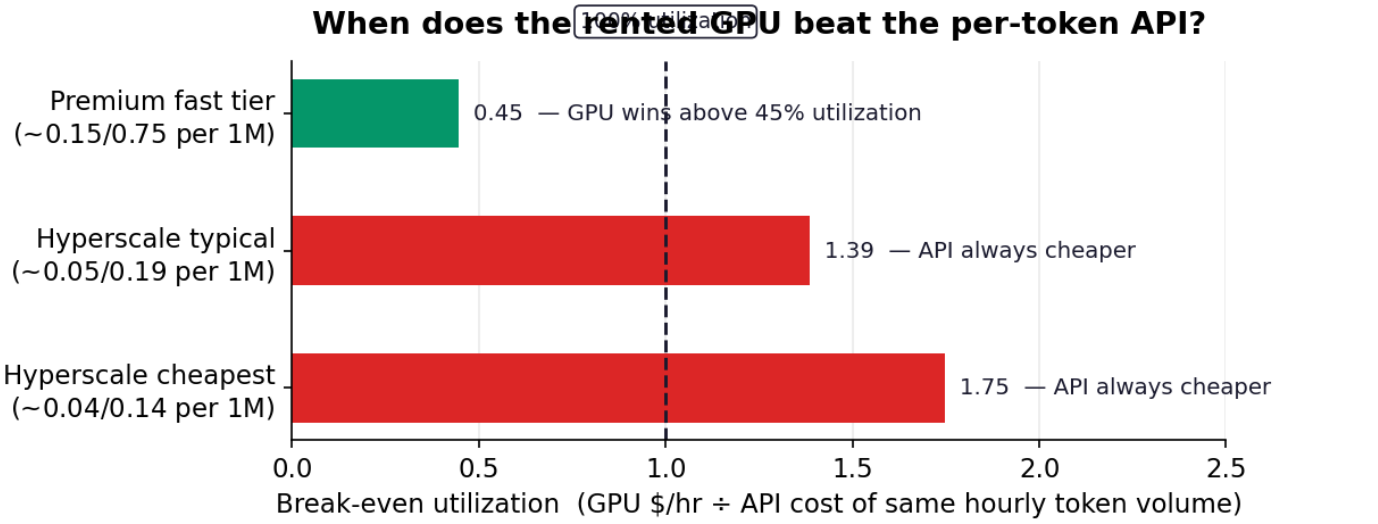
Gemma 4 E2B-it, 1×A100-40GB ($2.50/hr), identical configuration — only the traffic changed. 2.2× difference.

2. Bigger models are no longer proportionally more expensive to run. Our 50x-larger model, on newer hardware, cost the same per unit of work as the small one — and three times less per request. The instinct that "serious models need serious budgets" is increasingly outdated. What this means practically: capability upgrades that looked cost-prohibitive on last year's assumptions may already be affordable. The assumptions deserve a refresh, not a renewal.



Measured at maximum sustained load, unique-document traffic, July 2026. Hardware rented per hour.

3. Neither option wins outright — and the deciding number fits on a sticky note. The largest per-token providers pool demand from thousands of customers, achieving an efficiency no single company can match. Against their cheapest tiers, renting hardware never breaks even. But against the premium tiers — the ones sold on speed and reliability, at up to 8x the price — our rented machine broke even at 45% usage and delivered more consistent response times, with zero rate-limit interruptions and zero corrupted responses (the standard API route failed 7 times out of 300 in our comparison test). The deciding number is simple: *what your monthly AI volume would cost per token, divided into the hardware's rental price.* Your team can compute it in an hour. We published the full method, free, so they can.



GPT-OSS-120B on 1×B200 at $9/hr, measured saturation throughput (~119M tokens/hr, 98% input). July 2026 list prices.

What this means for your organization

Three questions for your next AI budget review:

1. Has anyone measured how repetitive our AI traffic actually is? (If the answer is no, you don't yet know what you should be paying.)
2. Are we buying premium per-token tiers for speed or reliability that dedicated hardware would deliver cheaper at our volumes?
3. What would our current monthly AI spend buy in rented hardware hours — and which side of break-even are we on?

There is also a strategic dimension money doesn't capture: running your own models means your data stays within infrastructure you control, your capacity can't be rate-limited during your busiest hour, and your supplier can't quietly change the model behind your product. Those risks rarely appear in cost comparisons. They appeared in ours.

The era when self-hosting AI required an infrastructure team is over — our entire evaluation was one command to deploy and $18 to run. The question is no longer whether your organization *can* compare the two options. It's whether it has.

Based on CognitX's July 2026 field benchmark: two open models, three GPU-hours, ~345,000 requests, full methodology and data published at [article].

Stop Comparing GPU Providers. Count Your Tokens.

Teams spend weeks comparing GPU clouds to host an open-source model. Most of them shouldn't be renting a GPU at all.

The comparison spreadsheet that feels like rigor

The scenario is common. A team picks an open model, say gpt-oss-120b, and starts shopping for a home. Someone opens the Hugging Face deploy menu and finds six options. Someone else builds a spreadsheet: dollars per GPU-hour on Modal, on RunPod, on Hugging Face Endpoints, on a bare-metal server rented by the month. The rows multiply. The meeting gets scheduled.

It feels like due diligence. Hourly rates are concrete, easy to rank, satisfying to compare. And the spread is real: the same H100 costs around $2 per hour on a reserved bare-metal cloud and up to $8 per hour on the most convenient managed endpoint.

But ranking GPU prices answers a question most teams never ask out loud: *should this workload be on a dedicated GPU in the first place?* Usually, it shouldn't. Flip the comparison.

Where an open model should run is decided by two numbers: how many million tokens you process per month, and how those tokens spread across the day. Not by which provider has the best pricing page.

Volume picks the tier. Timing picks the machine. Everything else is detail.

Why the spreadsheet wins anyway

Hardware prices are visible; usage patterns are not. A GPU rate fits in a cell. "How many tokens will our users actually generate in March, and at what hours?" requires estimation, and estimation feels like guessing. So teams optimize what they can see. It is rational behavior, and it routinely produces a $1,500-per-month server doing the work of a $40 API bill.

The three rungs

Below roughly 50 million tokens per month, renting hardware is a mistake at any price. Open models are served per-token by a dozen competing providers: DeepInfra, Together, Fireworks, Groq, or all of them at once through a router like OpenRouter. For gpt-oss-120b, input tokens start around $0.03 to $0.04 per million. A chat application serving a few hundred users burns single-digit dollars per month at those rates. Even the cheapest rentable GPU, a consumer RTX 4090 at about $132 per month, loses that comparison by an order of magnitude before you count the hours spent maintaining it. One caution from our own benchmarking: the same model varies up to 7.8x in price and dramatically in reliability across these providers, so pick one deliberately rather than routing blind.

Work that arrives in batches belongs on a GPU that dies when the job ends. A daily pipeline of 100,000 inferences does not need an always-on server; it needs two or three hours of compute, once a day. Rent a spot GPU on a marketplace like Vast.ai or RunPod for roughly a dollar an hour, run the batch through an inference engine, shut the machine down. The monthly bill lands between $30 and $90, below both the per-token equivalent and any standing infrastructure. This is the rung the spreadsheet never finds, because it isn't a provider. It's a schedule.

Steady, high-volume traffic is the only case for owning the serving stack. When interactive users keep a GPU busy most of the day (in practice, past 40 to 50% utilization, a threshold our own measurements put at 45%), the economics invert. A reserved H100 at about $2 per hour, running an optimized engine like vLLM, serves billions of tokens a month for a fixed ~$1,500. The same volume billed per-token costs a multiple of that. Public analyses place the crossover between 100 and 500 million tokens per month. Above it, the fixed cost stops being a burden and becomes the discount.

Four situations override the math entirely. If the model is your own fine-tune, per-token providers cannot serve it; you are in GPU territory regardless of volume. If data cannot leave your jurisdiction, you self-host even at a loss. If tail latency or output reliability carries a contract, dedicated tenancy wins on merit: in our field benchmark, a self-hosted endpoint produced zero malformed structured outputs across 344,000 requests, while one API provider on a routed endpoint failed on nearly a third of the same task. And if nobody on the team can own an inference engine, the managed premium (Hugging Face Endpoints, Friendli, Baseten) is not waste. It is the salary of the engineer you didn't hire.

Four questions before you rent anything

1. How many million tokens did you actually serve last month? Measured, not projected.
2. Do those tokens arrive all day, or in a burst you could schedule?
3. Could a per-token provider legally and technically serve your exact model?
4. Who on your team restarts the inference engine at 2 a.m.? If the answer is "nobody," which premium are you choosing to pay instead?

The provider list is long. The decision is two numbers. Count your tokens before you count your GPUs.

Leyton CognitX helps teams run exactly this arithmetic on their own workloads, before the infrastructure bill does it for them.

Sources

- gpt-oss-120b provider pricing and spread: Artificial Analysis, provider benchmark, July 2026 (https://artificialanalysis.ai/models/gpt-oss-120b/providers)
- Reserved H100 rates: Spheron GPU catalog / Lambda pricing, July 2026 (https://www.spheron.network/blog/modal-gpu-pricing-2026-per-second-billing/)
- Serverless vs dedicated break-even analyses, 30 to 61% utilization: Beam, Modal pricing breakdowns, June 2026 (https://www.beam.cloud/blog/modal-pricing-explained)
- Self-hosting inflection, 100 to 500M tokens/month: independent Hugging Face pricing analysis, June 2026 (https://techjacksolutions.com/ai-tools/hugging-face/hugging-face-pricing/)
- RTX 4090 monthly rental range: GetDeploying price tracker, July 2026 (https://getdeploying.com/gpus/nvidia-rtx-4090)
- Structured-output failure rates, 45% break-even, tail-latency findings: Leyton CognitX field benchmark, July 2026 (link to benchmark article/repo).

The Code Writes Itself. The Judgment Doesn't.

What AI Native DevCon 2026 revealed about the real bottleneck in agentic development

Five hundred builders spent two days at The Brewery in London (June 1–2) asking an uncomfortable question: what is software development, once agents do most of the typing?

The conventional story of the past eighteen months feels like success. The agent scaffolds a feature in minutes. The demo works. Leadership is impressed. Teams report individual productivity gains that would have sounded absurd in 2023. And that is exactly where most organizations have stopped.

Guy Podjarny, the Tessl founder who organizes the conference, framed the turn: 2025 was the year coding agents showed real promise; 2026 is the year we find out whether they hold up in production — across teams, codebases, and environments, without constant human correction. The useful question is no longer "can the agent do it?" It is "can we govern what the agent does, at the speed it does it?"

The bottleneck has moved from writing code to governing it — and the industry is now rebuilding the entire software stack around a new unit: the skill.

Four themes ran through the two days.

Skills are becoming the unit of software — with none of the infrastructure. A skill is a reusable, versioned set of instructions for an agent, and the thesis threading the conference was that these artifacts, not source files, are becoming what teams author, share, and depend on. The problem: we have recreated the early days of programming without any of its safety net. There is no mature static analysis for skills, no testing discipline (evals are the embryonic equivalent), no dependency management, no observability. Snyk's Liran Tal made the gap concrete in the best-titled talk of the event: "Your agent installed malware because a SKILL.md told it to." A skill is executable trust. Right now, almost nobody audits it.

Harness engineering is context delivery. OpenAI's Ryan Lopopolo argued that the binding constraints of software development have changed: they are now human time, human and model attention, and the context window. His prescription was to stop hoping the model infers your non-functional requirements and instead encode them into the harness — the review surfaces, approval gates, and context pipelines around the agent. Think of the harness as a factory jig: the craftsman's judgment, cast into tooling so every run inherits it.

Agents don't learn, so memory becomes architecture. Anthropic's Lamis Mukta named the quiet flaw in the agentic dream: intelligence doesn't compound. Task fifty starts as ignorant as task one. The industry's answer is a progression — from static instruction files to memory tools to skills to agent-managed memory, including out-of-band consolidation processes she described as "dreaming." The sharpest audience question of the conference cut through it: at what point are we reinventing databases from first principles? The honest answer seemed to be: we already are, and we should at least do it deliberately.

The human is now the congestion point. Thoughtworks' Birgitta Böckeler closed the event by naming the flow crisis: agents generate code faster than humans can review it, and the real costs sit beyond tokens — in the "harness tax" each tool imposes and in human energy. GitHub Next's Don Syme offered the industry a useful polarity: the hype lives in individual productivity, but the unsolved problem is team and SDLC continuity — what GitHub is betting on with agentic workflows and "continuous AI." Netlify's Dana Lawson added the design lens: platforms now need AX, agent experience, alongside UX and DX, because half the users of your APIs will soon be non-human. The scarce resource, she argued, is no longer typing speed. It is taste, judgment, and architecture.

The most grounded proof that this is not conference-circuit theorizing came from ReCinq and Odevo, a 14,000-person property management company restructuring itself as AI-native — with Meta's Ian Thomas describing the same reorganization at engineering-org scale.

For a team lead, the diagnostic writes itself. How many of the instructions your agents follow have been reviewed the way you review code? Can you name who authored the skills in your pipeline — and would you notice if one changed? When your agents double their output next quarter, what happens to your review queue? If any of these answers make you wince, the conference's message was for you.

Getting an agent to work is a demo. Getting a thousand runs to agree is engineering.

Based on AI Native DevCon 2026 (Tessl), The Brewery, London, June 1–2, 2026. All 40+ talks are available on demand at tessl.io/devcon.

The Models Got Smarter. We Didn't Change How We Talk to Them.

In 2020, GPT-3 could finish your sentence with something that sounded plausible. In 2026, OpenAI's Sol can coordinate a team of its own subagents across a multi-hour coding task, and Anthropic's Fable can hold its own against it on the hardest benchmarks in the field. That is a real leap: from a model that predicted the next word to a system that plans, delegates, and checks its own work.

Here is the part that doesn't get talked about enough. While the models were making that leap, most people's relationship with them barely moved.

A study OpenAI ran with Harvard economist David Deming, based on more than a million real conversations, found that nearly 80% of ChatGPT usage still falls into three buckets: writing, seeking information, and practical guidance. Coding, the exact frontier where Sol and Fable are racing each other on benchmarks, makes up about 4% of messages. For most users, a model built to run autonomous agentic workflows is being used the way people used Google in 2015: ask a question, get an answer, move on.

The real story of the GPT-3 to Sol/Fable era isn't that the models got smarter. It's that the gap between what they can do and what we ask them to do has never been wider.

That gap is worth sitting with, because it cuts against the usual narrative. The usual narrative says AI is racing ahead of us and we're scrambling to keep up, jobs disappearing, workflows being automated overnight. The data tells a quieter story. People are folding these models into daily life, but mostly at the surface. Non-work usage climbed from 53% to over 70% of all conversations in the space of a year, which says a lot about how personal this technology has become. It says much less about whether people are using it to do anything they couldn't do before.

A few shifts explain what actually changed, and what didn't.

Capability moved from single-shot text prediction to agentic execution. GPT-3 had no memory of its own actions and no ability to use tools. Sol's flagship mode can run parallel workstreams and act on results without a human in the loop at every step. That is a categorical change, not an incremental one.

Usage patterns moved much less than capability did. The same NBER study found the mix of writing, guidance, and information seeking has stayed remarkably stable even as the underlying model got dramatically more capable underneath it. People aren't rejecting the new capability. They mostly don't know it's there, or haven't found a reason to reach for it.

The growth that did happen was personal, not professional. More people are asking these models for advice, drafts, and explanations in their everyday lives. Fewer, proportionally, are handing over the kind of heavy, multi-step work these models are now actually built for.

None of this means people are using AI wrong. Writing help and quick answers are legitimate, valuable uses, and the consumer surplus from just that is measured in tens of billions of dollars. But it does mean the conversation about AI's impact has gotten the direction of the bottleneck backwards. We keep asking whether the models are ready for bigger tasks. The more useful question is whether we've noticed they already are.

The models did not stop evolving. We stopped asking new questions.