Hanzo Router: Memory-Aware, Local-First LLM Routing with a Learned SLO-Constrained Policy

Zach Kelling

Hanzo AI

Hanzo Research

Hanzo Router: Memory-Aware, Local-First LLM Routing with a Learned SLO-Constrained Policy

Zach Kelling*
Hanzo Industries Inc (Techstars '17)
Los Angeles, California
`https://hanzo.ai`

Hanzo Research

Abstract

Frontier language models are priced for their hardest queries, but a production request stream is dominated by its easiest ones: short chats, classification, formatting, and boilerplate code that a small local model answers as well as a frontier API. Paying the frontier price on every token is the single largest avoidable cost in an LLM product. We describe *Hanzo Router*, the request router that `hanzo-node` uses to place each request across a swappable pool of local and cloud models. The design decomplects *mechanism* from *brain*. The mechanism (`hanzo-router`) is pure logic over value inputs: a model registry, a memory snapshot, the set of already-loaded models, and a per-request service-level objective; it prefers, in order, reusing a model already resident (zero marginal cost), loading a local model that fits available memory, then the cheapest usable cloud model. The brain is a swappable policy behind a one-method seam: a rule-based cold-start policy, a learned bilinear utility with per-user online adaptation (`enso`), or a small trainable routing encoder (`zen-router`). We give the SLO-constrained objective the learned policy optimizes, quality $-\,\lambda\,\text{cost} - \mu\,\text{latency}$ under hard feasibility ceilings; the memory-aware fit check that makes local-first placement deterministic and unit-testable; the online LinUCB loop that personalizes routing from observed rewards; and a usage-plane signal that biases routing away from providers near their rate limits. We do not claim a measured benchmark saving. Instead we give a transparent, itemized cost model that goes beyond API token prices: for a mixed workload whose simple traffic is served locally for the price of electricity (not a rented GPU) and whose medium traffic drops to a cheap cloud tier, the arithmetic yields roughly a 90% reduction in total AI spend—token cost, avoided cloud-GPU rental, consolidated per-seat subscriptions, and rate-limit overage—in line with the published routing and cascade literature.

1 Introduction

A frontier language model is priced for the worst case. Its per-token rate has to cover the queries that genuinely need a trillion-parameter model: multi-step reasoning, long-context synthesis, hard code generation. But the request stream of a deployed product does not look like that benchmark. It is dominated by easy traffic: a one-line clarification, a yes/no classification, a JSON reformat, an autocomplete, a boilerplate function. On that traffic a 4B model running on a laptop is indistinguishable from a frontier API to the end user, and it costs only electricity per token—about two orders of magnitude below frontier API pricing (§6)—on hardware already paid for.

Routing every request to the frontier model therefore spends the frontier price on a stream whose median query does not need it. This is the single largest avoidable line item in an LLM product's bill, and it is avoidable precisely because the mismatch is structural: the price is

set by the tail, the volume is in the head. A router that sends each request to the cheapest model that can still serve it converts that structural mismatch into a cost saving, without a user-visible quality regression, provided it can tell the two kinds of request apart and provided it has somewhere cheap to send the easy ones.

This paper describes the router Hanzo runs in production, `hanzo-router`, and its learned policy `enso`. The contribution is not a new model; it is an architecture that makes cheap-when-you-can, frontier-when-you-must routing a first-class, testable, learnable component of the serving stack. We make four claims. First, the routing *mechanism* should be pure logic over value inputs, with no hardware access of its own, so that placement is deterministic and unit-testable (§2). Second, local-first placement is a memory-fit problem: given a snapshot of available memory and the set of models already resident, the router reuses, loads, or falls through to cloud by a single ordered walk of preferences (§2). Third, the *policy* that ranks models is a swappable brain behind a one-method seam, so a rule guess, a learned bilinear utility, and a trainable encoder are interchangeable and compose (§3, §4). Fourth, the saving is real but must be stated honestly: we give a transparent cost model (§6) rather than a fabricated benchmark, and anchor the magnitude in the published routing literature (§7).

2 The Mechanism: Memory-Aware, Local-First Placement

The mechanism is the crate `hanzo-router`. It is deliberately dumb about hardware and clever about values. Following the Rich Hickey discipline of separating a decision from the place it runs [4], `hanzo-node` gathers the facts the decision needs into plain values and hands them to a pure function; the router touches no device and opens no socket, so the same inputs always produce the same decision and every path is a unit test.

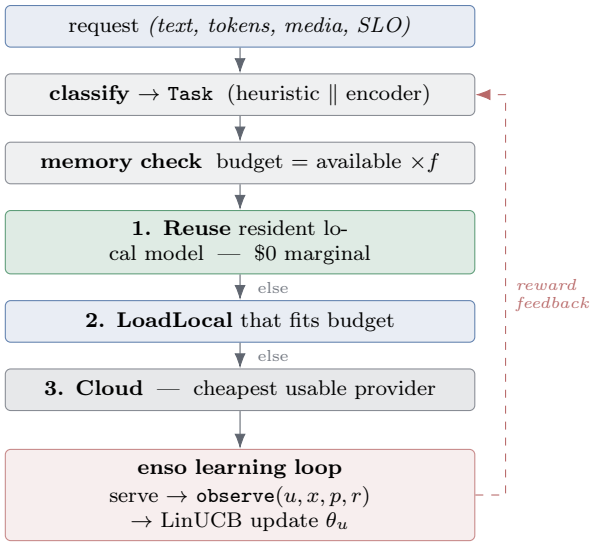

Figure 1: The router decision flow. A request is classified to a task, checked against the memory budget, and placed by the ordered walk: reuse a resident local model (zero marginal cost), else load a local model that fits, else fall through to the cheapest usable cloud model. The learned policy (`enso`) closes the loop—each served request yields a reward that updates the per-user LinUCB weights and, through them, future routing.

2.1 The pool as a registry of value rows

Every servable model is one `ModelCard`: a canonical id, a `Backend` that is either `Local{est_bytes}` (served by a local `hanzo-engine`, with the resident footprint the router fits against memory) or `Cloud{provider}`; the set of `Task`s the model is preferred for; its maximum context; a vision flag; and a relative `cost_per_1k`. Adding a model is adding a row. Tasks are a coarse taxonomy—`Code`, `Reasoning`, `Math`, `Creative`, `Vision`, `LongContext`, `CheapChat`, `General`—used only to rank quality within a modality pool; modality (text, vision, image, audio, video) selects the pool itself. The two axes are orthogonal, so one router serves chat, vision, and media generation by matching pools.

A request is classified to a task by a `Classifier`. The default `Heuristic` is keyword-, length-, and media-based: fenced code or words like "bug" and "refactor" route to `Code`; "prove", "theorem", "in-

tegral" route to `Math`; a prompt over 32k tokens is `LongContext`; a prompt of a handful of tokens is `CheapChat`; everything else is `General`. The heuristic is a seam, not a commitment: the `Classifier` trait lets a learned model produce the same `Task` value without the policy changing, which is exactly where `zen-router` (§3) plugs in.

2.2 Memory as a value, and the fit check

The one physical fact the router needs is how much memory it can spend. That, too, is a value: a `MemSnapshot` carrying `available_bytes`, `total_bytes`, and a `unified` flag, filled by `hanzo-node` from the same `MemoryUsage`/`DeviceMemory` signal the engine's auto device-mapper reads. The usable budget for resident weights is a fraction of available memory,

$$\text{budget} = \text{available\_bytes} \times f, \qquad (1)$$

and a local model of footprint b fits iff $b \leq$ budget. The default fraction mirrors the engine's own policy: $f = 0.85$ on unified-memory devices (Apple Silicon, GB10/DGX-Spark, Strix Halo), where weights, KV cache, and activations share one pool, and $f = 0.7$ on a discrete GPU, where a reserve is kept for activations and KV. The operator can override f; nothing else about the fit check is tunable, because nothing else needs to be.

2.3 The ordered walk

Given a classified task, the registry, a memory snapshot, and the set of currently-running model ids, the decision (`Policy::select`) is a single ordered walk of the task's candidate models. Candidates are assembled in preference order: the policy's explicit `prefer[task]` list first, then models that explicitly advertise the task, then `General` catch-all models last—so a task specialist outranks a generalist even if the specialist is in the cloud. The walk then applies one rule:

1. **Reuse.** If any usable candidate is local and already resident in a connectable engine, serve it. This is the zero-marginal-cost path and it wins outright, even over a smaller model that would also fit, because a load already paid for is cheaper than any load not yet paid for.

2. **Load local, or fall through.** Otherwise walk the candidates in preference order. The first that is a local model whose footprint *fits* available memory is loaded and served; the first that is a usable cloud model is routed to. A higher-preference local model that does not fit is skipped, and the walk falls through to the next preference—which may be cloud. This is literally "run it locally if the RAM is there, else the next best wherever it is."

3. **NoFit.** If nothing is resident, nothing local fits, and no cloud model is usable, the router returns `NoFit` and the caller errors rather than silently degrading.

Usability filters a candidate out before it is considered: a vision request requires `vision`; a request needing k tokens of context requires `max_context` $\geq k$; and a cloud model priced above the policy's `cost_ceiling` is refused. The whole of `select` is a few dozen lines and every branch above is a named test in the crate—reuse-over-reload, fall-to-cloud-when-local-does-not-fit, vision-requires-a-vision-model, cheap-chat-prefers-cheapest-cloud, and no-fit.

3 The Brain: A Swappable Policy Behind One Seam

Placement is mechanism; *which model is best for this request and this user* is policy, and policy is where learning belongs. The two are joined by a single trait, `RoutePolicy`, with one method: given a request, the user it is for, an SLO, and the registry, return a `Route`—a model, an effort `Level` (`Fast`/`Balanced`/`Max`), a modality, and a confidence. Everything the mechanism owns—the registry, the SLO gate, placement, dispatch—sits on one side of the seam; everything the brain owns sits on the other. Three brains implement the seam and interchange freely.

The rule-based cold-start policy. `hanzo-router`'s own `Policy` implements `RoutePolicy`: it classifies with the heuristic, walks candidates in preference order, and returns the first usable one at `Balanced` with a fixed low confidence—enough to act on, low enough that a learned policy knows it is an un-personalized guess. This is the floor: a fresh deployment routes sensibly before any eval data exists, and any learned brain delegates to it when it has no profiles for a pool.

The learned utility (`enso`). The production brain is `enso`, six orthogonal pieces with one concern each: a featurizer (request $\to$ feature vector x), a profile table (eval rows p per (model, level, modality)), a learnable bilinear utility, a two-tier safety guard, an SLO-constrained selector, and a learner (§4). It scores every candidate profile by a bilinear form and picks the SLO-feasible argmax, falling back to the rule policy on cold start or an empty pool.

The trainable encoder (`zen-router`). At the classifier and featurizer seams sits an optional 0.6B routing model, `zen-router` [12], with task, route, and feature heads. It is supervised from eval and usage-ledger data and refined with GRPO [9] on the same objective the selector optimizes, quality $-$ λ cost $-$ μ latency. Because it produces exactly the `Task` and feature values the existing seams already consume, it drops in without touching the mechanism or the selector—a learned front-end for a policy whose shape does not change.

4 The SLO Objective and Online Adaptation

4.1 Bilinear utility

`enso`'s model of quality is deliberately a single matrix. For a request feature vector $x \in \mathbb{R}^D$ and a profile vector $p \in \mathbb{R}^K$, the utility is the bilinear form

$$u(x, p) = x^\top W p, \qquad W \in \mathbb{R}^{D \times K}. \tag{2}$$

With the task one-hot in x and quality-by-task in p, W learns exactly the task-to-quality alignment existing needs, and stays one matrix an online learner can update convexly. A richer head—an MLP over $\phi = \mathrm{vec}(xp^\top)$, a low-rank neural encoder—attaches at this same seam by replacing the dot with a network, without touching any other piece; the linear head is what keeps the online learner convex and the offline fit closed-form.

4.2 The SLO-constrained decision

The per-request service-level objective is a value the caller sets—the operator's budget for *this* request, distinct from the user's learned taste. It carries hard ceilings (`max_latency_ms`, `max_cost`, `min_quality`; zero disables a ceiling) and two soft weights (λ = `lambda_cost`, μ = `mu_latency`). Over the profiles in the request's modality pool that satisfy the ceilings, the selector takes

$$\arg \max_{p\,:\,\mathrm{feasible}(p)} u(x, p) - \lambda\, \widehat{c}(p) - \mu\, \widehat{\ell}(p), \tag{3}$$

where $\widehat{c}$ and $\widehat{\ell}$ are normalized cost and latency and feasibility is

$$\ell(p) \le \texttt{max\_latency\_ms} \wedge c(p) \le \texttt{max\_cost} \wedge q_t(p) \ge \texttt{min\_quali} \tag{4}$$

The hard ceilings gate; the soft weights trade. A cost-saver deployment raises λ; a latency-first deployment raises μ; a quality floor is enforced, not traded, by `min_quality`. Confidence is the squashed margin to the runner-up, so a close call is reported as a close call. The same `feasible` predicate gates the learned selector and the ground-truth oracle, so the two are compared on identical constraints.

4.3 Online adaptation

Offline, the base W is a closed-form ridge fit [5] of observed eval quality on the bilinear feature $\phi = \mathrm{vec}(xp^\top)$. Online, each user carries a contextual bandit—LinUCB [6]—whose parameter θ_u starts at a Gaussian prior centered on the base W and drifts toward that user's taste as outcomes arrive. The serving hot path is greedy on θ_u; the exploration bonus is computed only while learning. The feedback is one call,

$$\texttt{observe}(u, x, p, r), \tag{5}$$

recording a reward r for the chosen (model, level, modality); the Sherman–Morrison update keeps A^{-1} and θ_u current in closed form, and $\|\theta_u - W\|$ is the readable size of a user's personalization. Because the reward is linear in ϕ, the LinUCB core is realizable and converges. Research-frontier variants—a low-rank neural ΔW_u, self-adaptive expert vectors—attach at exactly this update seam and are deliberately not faked in the shipped code.

5 Usage-Plane Signals as Decode Masks

Cost and memory are not the only constraints on placement; provider rate limits are a third. Hanzo's usage plane (`@hanzo/usage`) tracks, per provider, the current rate-limit window and spend against it. A snapshot of that state is a value the router can read like any other: a provider near its requests- or tokens-per-minute ceiling is temporarily down-weighted or removed from the usable set, exactly as an over-budget cloud model is removed by the cost ceiling. Framed against the learned policy, a saturating provider acts as a decode mask over the candidate set—it does not change what the ideal model is, it changes which models are available to serve right now—so routing degrades gracefully around a throttled provider instead of failing requests against it. The signal is advisory and orthogonal: it composes with the memory fit check and the SLO gate rather than replacing either.

6 Cost Model

We do not report a measured benchmark saving, because we have not run a controlled A/B at the scale that would make such a number honest. What we can do is state the arithmetic transparently, so the reader can substitute their own workload mix and prices and see the saving fall out. The saving is not a property of any model; it is a property of the workload's task distribution against a price schedule.

Segment	Share	Routed to	$/req	Routed $
Simple	70%	local (zen)	0.00000	0
Medium	20%	cheap cloud	0.00033	66
Hard	10%	frontier	0.00750	750
Routed total				**816**
All-frontier	100%	frontier	0.00750	**7,500**

Table 1: Per-token API spend per one million requests. Baseline routes everything to the frontier model ($7,500); the router serves simple traffic locally (no per-token API charge), drops medium traffic to a cheap tier, and reserves the frontier model for the 10% that needs it ($816). Prices and mix are inputs, not measurements; substitute your own.

6.1 Per-request token economics

Take a stream of one million requests, each averaging 1,000 input and 300 output tokens. A frontier model priced at $3.00 per million input tokens and $15.00 per million output tokens costs, per request, $\frac{1000}{10^6}\cdot 3.00 + \frac{300}{10^6}\cdot 15.00 = \$0.0075$. A cheap cloud tier priced at $0.15/$0.60 per million costs $0.00033 per request—about 1/23 of the frontier rate. A local zen model served by `hanzo-engine` carries *no per-token API charge*: the marginal API cost is $0 (its electricity cost is a compute line, accounted separately in §6.2). Segment the stream by what each request needs: 70% simple (chat, classification, formatting) → local; 20% medium (general, routine code) → cheap cloud; 10% hard (reasoning, long context) → frontier. Table 1 works the API spend.

The routed stream costs $816 in API spend against $7,500 all-frontier, a $1-816/7500 = 89.1\%$ reduction. The "up to 90%" headline is exactly this arithmetic, and it moves with the mix as expected: at an 80/15/5 split the saving is 94%, and a stream with no simple traffic saves nothing. The lever is the share of requests that do not need the frontier model, which in real assistant and agent traffic is the majority.

6.2 Compute cost: local is cheap, not free

Serving locally is not free—it draws power on hardware that is already amortized. A small model (0.6B–24B) on consumer hardware draws on the order of 50 W sustained and decodes roughly 25 tokens/s, i.e. about 2 J per token. At the U.S. average residential rate of ≈ \$0.15/kWh [10], a request of ∼1.3k tokens costs

$$\frac{2\,\text{J}\times 1300}{3.6\times 10^6\,\text{J/kWh}} \times \$0.15 \approx \$0.0001,$$

about one one-hundredth of a cent, and roughly two orders of magnitude below the frontier API's ∼ \$0.006/1k blended. These are order-of-magnitude estimates (device wattage, throughput, and tariff are all inputs, not measurements); the point is that the local token bill effectively disappears and the residual cost is the *amortized hardware*, which the developer's machine already exists to provide.

Versus rented GPU. The alternative way to get zero-per-token economics is to self-host the open model on a *rented* cloud GPU. One always-on entry-GPU instance at ≈ \$0.80/hr is ≈ \$580 per month; a redundant pair doubles it. Local-first routing onto laptops and workstations the team already owns avoids that rental entirely—the compute the organization has already bought becomes the local tier.

The fleet angle. Owned compute can be pooled. A device that registers into Hanzo Cloud as a run-for-pay node runs the open `hanzoai/cloud` binary and serves an organization's `/v1` traffic through the same metering plane, turning idle compute into capacity instead of buying more cloud [3]. Served work is priced cost-plus a fixed resell margin (25% over measured wholesale) and metered by the commerce ledger, and 25% of that served revenue is paid to the OSS contributors whose code ran it. For an organization, the practical effect is that spare capacity on the fleet offsets cloud spend rather than sitting idle.

Monthly cost line (20-person team)	Status quo	Router
API token spend (Table 1)	7,500	816
Local-tier compute (electricity)	0	70
Per-seat assistant subs (20 × \$20)	400	0
Provider overage / burst tiers	150	0
Routing + metering fee (1% of routed usage)	0	8
Monthly total (metered bill)	**8,050**	**894**

Table 2: Total monthly AI cost for a 20-person team on the Table 1 workload. The metered bill falls from \$8,050 to \$894, an ≈ 88.9% reduction. *Additionally, not summed above:* to run the local tier on a rented GPU instead of owned hardware would cost ≈ \$580 per month—avoided entirely by local-first placement (the status-quo column bills frontier API, not self-hosting, so the rental is a separate avoided alternative). All figures are labeled assumptions, not measurements.

6.3 Total cost of ownership: a 20-person team

API tokens are one line of the bill. Table 2 totals the monthly cost of a 20-person engineering team running the one-million-request workload above, against the status quo of a frontier-default API plus per-seat assistant subscriptions. Beyond tokens, three further costs move: *per-seat subscriptions* collapse when org login shares one metered account through the usage plane instead of 20 individual seats; *provider overage* disappears when rate-limit-aware routing shifts load before a window exhausts, so no burst/overflow tier or wasted reset credit is bought; and *operations* simplify to one metering/billing plane (BillingGate over the commerce ledger) instead of reconciling a stack of per-provider invoices. The consolidation itself is priced as a small platform fee on routed usage—illustratively 1%—which we include as a cost so the total is not flattered.

The metered bill drops from \$8,050 to \$894, an ≈ 88.9% reduction, and local-first additionally avoids the ≈ \$580 per month a rented GPU would cost to serve the same local tier. The headline is the same as Table 1's 89.1% because API tokens dominate the bill; the point of the itemization is that the other lines—compute, seats, overage, ops—move in the *same* direction, so the

total saving is not an artifact of token prices alone. This is the economics the cascade and routing literature reports (§7); our contribution is the mechanism that realizes it deterministically and the policy that learns where the line between "simple" and "hard" actually falls.

7 Related Work

LLM routing and cascading are an active area, and the cost economics above are consistent with its published results. FrugalGPT [1] cascades from cheap to expensive models with a learned scorer that decides when a cheap answer suffices, reporting large cost reductions at matched quality. RouteLLM learns a binary strong/weak router from preference data and reports frontier-level quality at a fraction of the cost by sending only the queries that need it to the strong model [8]. Hybrid LLM [2] routes between a small and a large model on a quality-aware threshold, and AutoMix [7] mixes models with self-verification to decide escalation. Tabi [11] is a multi-level inference system that serves small models first and escalates selectively. Our objective, quality$-\lambda\,$cost$-\mu\,$latency, is the natural continuous generalization of these binary escalation rules to a pool of many models across local and cloud backends, and the per-user LinUCB loop [6] adds online personalization that cascade work generally does not. What is distinctive here is not the routing idea but the engineering posture: a mechanism that is pure over value inputs and memory-aware, so that local-first placement is deterministic and testable, and a policy seam that lets a rule, a bilinear utility, and a trainable encoder interchange without a rewrite.

8 Limitations

The honest scope is narrow in three places. First, the profile table that the selector ranks over is populated from real eval data when a bench's JSONL is available and from clearly-labeled synthetic tuples otherwise; until per-pool eval data lands, the learned policy's absolute rankings are only as good as those synthetic profiles, which is why the rule-based cold-start policy remains the floor. Second, aggressive downgrading risks a quality regression on requests the classifier misjudges as simple; the `min_quality` ceiling is the guard—it is enforced, not traded, so a request with a quality floor is never served by a model below it, and a misclassified-hard request is caught by the floor rather than by the soft cost weight. Third, the cost model is arithmetic, not a measurement: it shows the shape of the saving under stated prices and a stated mix, and a deployment's real saving depends on its own traffic. We report the model rather than a benchmark precisely so that the claim cannot be mistaken for one we did not run.

9 Conclusion

Frontier pricing is set by the hardest query; production volume is in the easiest. Hanzo Router turns that structural mismatch into a cost saving by placing each request on the cheapest model that can still serve it: reuse a resident local model for free, load a local model that fits memory, and reserve the cloud—and the frontier model within it—for the traffic that needs it. The mechanism is pure logic over values and memory-aware, so placement is deterministic and unit-tested; the brain is a swappable policy behind one seam, so a rule guess, a learned bilinear utility with online per-user adaptation, and a trainable routing encoder interchange and compose. The SLO objective quality $-\,\lambda\,$cost $-\,\mu\,$latency makes the cost/quality/latency trade explicit and per-request, with hard ceilings that gate and a quality floor that is enforced rather than bargained away. The saving is not magic and we do not dress it as a benchmark: it is the arithmetic of not paying the frontier price on a stream whose median request never needed it.

References

[1] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. *arXiv preprint arXiv:2305.05176*, 2023.

[2] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-Efficient and Quality-Aware Query Routing. In *International Conference on Learning Representations (ICLR)*, 2024.

[3] Hanzo Industries. Hanzo Cloud Open Edition: Run-for-Pay Pricing and Contributor Revenue. Hanzo architecture note, `https://github.com/hanzoai/cloud`, 2026.

[4] Rich Hickey. Simple Made Easy. Strange Loop Conference, 2011. `https://www.infoq.com/presentations/Simple-Made-Easy/`.

[5] Arthur E. Hoerl and Robert W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12(1):55–67, 1970.

[6] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A Contextual-Bandit Approach to Personalized News Article Recommendation. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 661–670. ACM, 2010.

[7] Aman Madaan, Pranjal Aggarwal, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Mausam, and Manaal Faruqui. AutoMix: Automatically Mixing Language Models. *arXiv preprint arXiv:2310.12963*, 2023.

[8] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. RouteLLM: Learning to Route LLMs with Preference Data. arXiv preprint arXiv:2406.18665, 2024.

[9] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300*, 2024.

[10] U.S. Energy Information Administration. Electric Power Monthly: Average Price of Electricity to Ultimate Customers by End-Use Sector. `https://www.eia.gov/electricity/monthly/`, 2025.

[11] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. Tabi: An Efficient Multi-Level Inference System for Large Language Models. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, pages 233–248. ACM, 2023.

[12] Zen LM. zen-router: A 0.6B Trainable Routing Model. `https://github.com/zenlm/zen-router`, 2026.