THALAMUS — Which AI Should I Ask?

Oscar Serra · Independent Research

30 September 2026

Abstract

Imagine you have a secretary, a very smart one. However smart she is, she cannot answer every question herself, and she is smart enough to know who can. The proof goes to a mathematician, the letter to a careful writer, the log of a break-in to someone who will read it instead of refusing. If one of them is busy, she calls the next, and the work never stalls. She also knows which binder of old procedures fits the job, even one filed under something else. Deep in your brain, the thalamus does much the same: it passes each incoming signal to the part of the cortex built to handle it.

AI models differ in intelligence and price, in how hard they think, in what they do best, in how much they can read at once and in which topics they refuse, and sending each task to the model that suits it can beat every model in the pool. But a task given to an AI assistant is not one question. It is a chain of model calls: plan, look something up, read the result, write, check. Each call has its own best model, yet routers choose once per task and keep that model to the end, and calls that could run at the same time run one after another.

We propose Thalamus, a router that chooses at every call: which model, how hard it thinks, how much of the conversation it gets to see, and what runs beside it. At each decision point Jev, a small model that answers only in typed choices, describes the situation in under half a second, and the router prices every option in money, quota, cache and time at the setting of one dial from fast to smart. Nothing is trained.

The same read ranks the assistant's own enhancements, its skills, recipes and plugins, including ones built for another job, and hands the agent a short list in order of probability. A loop watches which one the agent actually uses and rewrites the descriptions Jev reads, so the next list is a little better.

In your own assistant, a long document is read in pieces at the same time, a long search result is condensed by a cheap reader instead of being carried to the end of the task, a stuck step moves up to a stronger model, the check goes to a model from another family, and the letter is written by the model that writes that language best. We show why the design moves pieces of work instead of whole conversations: a model that has been reading a conversation re-reads it from its cache for a tenth of the price or less, so handing the whole thread to a cheaper model pays only after several steps. This is a design, and §10 names the tests that would prove it.

If it holds, an assistant stops being one model doing everything, and every call reaches the one who does it best.

1. Introduction

Picture four requests arriving in the same assistant on one afternoon. The first is a proof in group theory. The second is a short letter in Swahili. The third is the log of a break-in, full of real exploit code. The fourth is a 600,000-token contract to search for early-termination clauses (a token is about three quarters of an English word). On that afternoon one provider's five-hour allowance is used up, and another has three quarters of its week still unused.

No single model is the right answer to all four. The proof wants the strongest reasoner. The letter wants a model that actually writes good Swahili, which is a different thing and need not be the same model: language coverage is not depth of thought, and here the cheaper model may well write the better letter. The log wants a model that will read it at all. When Hugging Face was broken into in July 2026, its responders first sent the attack logs to hosted frontier models, whose safety guardrails blocked the work, and then ran the forensics on GLM-5.2, an open-weight model from Z.ai, on their own machines. "The attacker was bound by no usage policy," their disclosure says, "while our own forensic work was blocked by the guardrails of the hosted models we first tried" (Hugging Face, 2026). The contract does not fit in some models at all, and the ones it fits read it in one long pass. All of it has to happen inside quotas that refill on their own clocks.

A router that picks the right model for each request already beats any fixed choice. Now look inside one of those requests. Searching the contract is not one question to one model. The assistant plans the search, reads the document, notes each clause it finds, checks the doubtful ones and writes the answer: dozens of calls in a chain. The plan wants a strong reasoner. Reading 600,000 tokens for one kind of clause is mechanical work, and it splits into pieces that can be read at the same time. The check wants a model from a different family from the one that found the clause. The answer wants a good writer. A router that chooses once per request picks one of these needs and pays for it on every call, and it runs the pieces one after another.

So the question changes from "which model for this request?" to "which model for this call, with how much of the conversation, and alongside what?" Answering it is hard for five reasons.

- There is no single ranking. A model's value depends on the kind of work, the effort it is asked to spend and the price you pay, and one request holds work of several kinds.
- The ground moves. Models arrive and are re-priced every week, and benchmark scores are rescaled.
- Quotas run on clocks. Subscription plans meter use in five-hour, weekly and monthly windows, so the same call is dearer when a window is nearly spent than when it is about to refill unused.
- Switching has a price. A model that has been reading a long conversation keeps it in a cache and reads it again for a tenth of the normal price, or less. A new model starts cold and pays in full. Switch at the wrong moment and the cheap model costs more than the expensive one.
- A chain is slow. Each call waits for the one before it and re-reads the conversation first. The number of calls in a line sets how long a task takes more than the speed of any one model does.

Models are not the only thing to choose. An assistant also collects enhancements: skills, recipes and plugins that hold the lessons of earlier work. Agents often fail to reach for the one that fits, and they rarely see that one built for another job would fit this one.

We propose a design for Thalamus that answers all five with one decision, taken at every model call (Figure 1), and hands the agent a ranked short list of its enhancements.

1. Where a model can change (§5): the points at which a harness can hand work to another model, from the whole request down to one call, what each hand-over carries and what it loses, and a rule for when it pays. A switch of the whole conversation pays only for a run of more than N^* steps, where N^* depends on the prices of cached and uncached reading and not on the length of the conversation.
2. Finishing sooner (§6): a scheduler that plans a task as a graph, starts every unit whose inputs are ready, gives the critical path the fast models and everything else the cheap ones, and covers slow calls with a second copy.
3. A typed reader at every decision point (§4): Jev describes each task and each step with calibrated confidence. The reader describes; the prices decide.
4. One price per option (§3): money, quota pace, cache state and time on the critical path in a single number, at the setting of a single dial.
5. The right enhancement (§7): Jev ranks the assistant's skills, recipes and plugins for each task, including ones that fit by their structure and not their purpose, and the agent picks from a short list in order of probability. A nightly loop watches what the agent uses and rewrites the text Jev reads.
6. The whole design (§8), traced on the four requests (§9), with the tests that would prove it (§10).

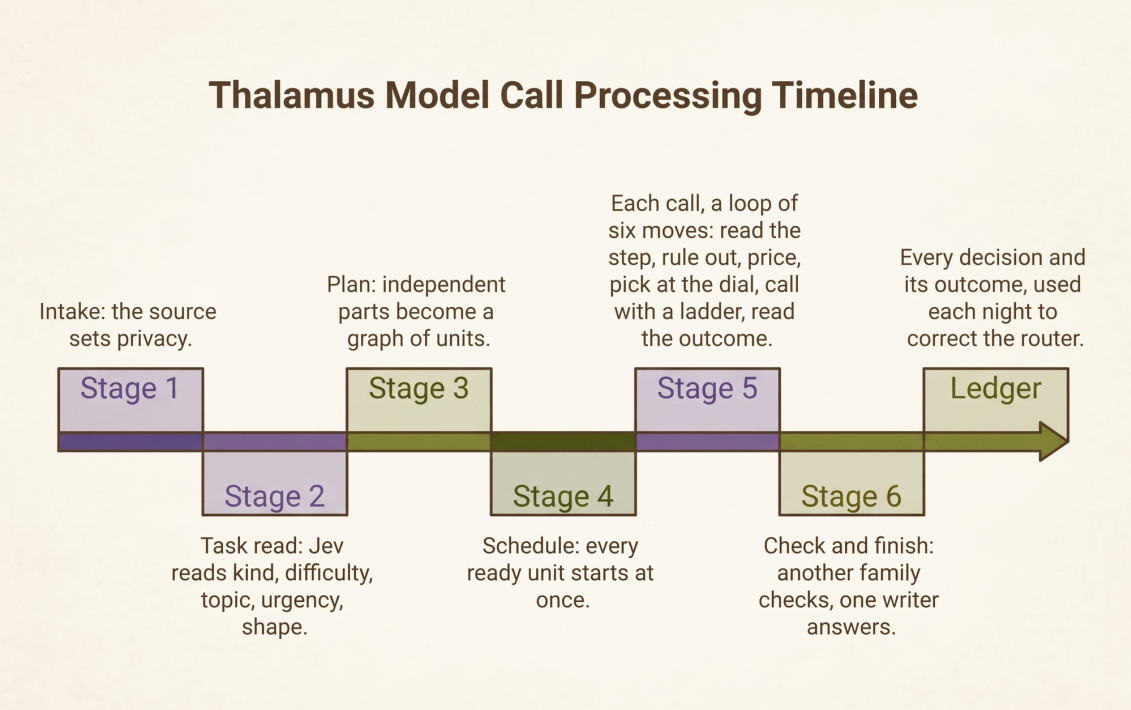


Figure 1. Thalamus at every call. The source of a request settles its privacy before anything leaves the machine; Jev reads the task; a request made of independent parts becomes a graph of units, and every ready unit starts at once. Inside a unit, every call repeats six moves: Jev reads the step while the previous tool runs, the vetoes rule options out, each remaining option is priced, the dial picks, the call goes out with a ladder behind it, and Jev reads the outcome. Work that commits or claims is checked by another family, and one writer finishes. The ledger, drawn last on the line, records every decision along the way (§8).

2. Related work

Fugu (Tang et al., 2026) is the work this paper follows. It routes and composes frontier models: its single-worker version picks one model per query, and its composing version, Fugu-Ultra, builds a workflow of up to five steps in which each step sees earlier steps only through a declared list. Fugu learns which model is good at what in a training run. Thalamus keeps the idea, route by what each model does best and compose only when it pays, and replaces the training with public measurements, a typed reader and a ledger of its own outcomes (§3, §8). It also routes below the request, at each call (§5).

RouteLLM (Ong et al., 2024) learns from preference data when a cheaper model is good enough and when to call a stronger one, and **FrugalGPT** (Chen, Zaharia and Zou, 2023) tries a cheap model first and escalates when the answer looks weak. Both choose for a whole query. Thalamus chooses among many models, each at several efforts, for each call, and escalates when a step is stuck (§5.4).

Adaptive routing under budget constraints (Panda et al., 2025) combines a prior from offline data, feedback from use and a budget, and is the nearest relative of the ledger's learning (§8). The budgets here are subscription windows with clocks of their own (§3).

Mixture-of-Agents (Wang, J. et al., 2024) layers models so that each reads the previous layer's answers. **Self-MoA** (Li, W. et al., 2025) found that sampling the single best model several times often beats mixing different ones. Thalamus puts several models on one question only at the top of the dial, when the leader's margin is small (§6.4).

LiteLLM (2026), a production gateway, falls back across deployments on rate limits, over-long requests and content-policy refusals. Its auto-router asks Jev one question, how complex a request is, and picks from a fixed pool for that tier. Thalamus asks Jev at every call, several questions at once, and chooses from measured strengths and live prices (§4).

Choosing tools. Gorilla (Patil et al., 2023) and ToolLLM (Qin et al., 2023) retrieve the right API out of thousands before a model calls it, with retrievers trained for the purpose. Thalamus ranks larger units, whole procedures and toolsets, with a typed reader and no training, and it looks for fits by structure as well as by topic (§7).

Learning from biased picks. Unbiased learning to rank (Joachims, Swaminathan and Schnabel, 2017) corrects click data for the pull of the top position, and §7.3 applies it to an agent's picks. Automatic prompt optimisation (Pryzant et al., 2023) and DSPy (Khattab et al., 2023) improve a prompt from the cases a model gets wrong, scored on held-out data; §7.4 does the same to the text Jev reads.

Faster answers. Skeleton-of-Thought (Ning et al., 2023) writes the outline of a list-shaped answer and fills the parts at the same time. Speculative decoding (Leviathan, Kalman and Matias, 2023) lets a small model draft tokens that a large one then checks. Hedged requests (Dean and Barroso, 2013) send a second copy of a slow request and keep whichever answer arrives first. Speculative decoding happens inside one vendor's servers, below anything a harness can reach; the other two become tools of the scheduler (§6).

J11 AMYGDALA (Serra, 2026a), the companion paper, asks Jev at every step whether an agent has misread its instructions or is about to do something it cannot undo. Thalamus asks its routing questions in the same call and writes to the same ledger (§4). The plan graph of §6 is the kind of workflow J13 PREFRONTAL runs (Serra, 2026b).

The map rests on public sources: the Artificial Analysis Intelligence Index (Artificial Analysis, 2026) for general capability, the Epoch AI benchmarking hub (Epoch AI, 2026) for strength in each kind of work, and OckBench (2025) for how many tokens a model writes per task at each effort.

3. What the router knows

Before it can choose, the router has to know what each option is worth and what it costs. It keeps six kinds of knowledge, each with its own clock.

The map. Each model appears once for every thinking effort its vendor offers, and we call each such point a *rung*. A rung has a height, the model's general intelligence at that effort, and a width, the cost of one typical task's output. A rung is on the *frontier* when no other rung is both cheaper and at least as smart. Beside the heights sits a table of strength in each kind of work (code, tool-using agents, mathematics, writing, languages, long documents and so on), each a percentile across the public benchmarks for that kind of work. Some kinds of work have no public benchmark at all, and there the router has no measured reason to prefer one model until its own ledger gives it one (§8). Models and prices change every week, so the map is rebuilt every day.

Two prices. A plan and a price list answer different questions. The list price is what a model costs anyone. The plan price spreads a subscription's fee over the tokens the plan serves, which is what a token costs the plan's owner on average. The router uses the plan price where a plan applies and the list price elsewhere.

Quota pace. For each quota window the router compares the share already used with the share of time already gone. A window spent faster than it refills makes that provider dearer; one spent slower makes it cheaper, because under a subscription whatever is left at the reset is lost. A short window with room to spare does not make a provider cheap when a longer window of the same account is nearly spent, since every token counts against both. A spent window is a veto until it refills. Credit bought by the token never gets cheaper this way, because it does not expire.

Cache state. Vendors keep the start of a conversation in a cache for five minutes or an hour, and a model that reads a cached start pays a fraction of the normal input price for it: a tenth on most models, a twentieth on Claude Opus 5.5 and a fortieth on Claude Fable 5.1. Writing the start into the cache costs 1.25 times the normal price for five minutes and twice for an hour (Anthropic, 2026). The cache holds the model's own internal state for those tokens, so no other model can use it. The router keeps a *cache ledger*: for each model, how much of the current conversation is warm and until when. Every response reports its cached and uncached tokens, so the ledger is kept from counts, not guesses.

Time. For each rung: how long before the first token, how fast it writes, and how long it usually thinks at that effort. A long uncached start delays the first token, and a cached one shortens the wait.

The dial. The operator sets one dial with named stops from fast to smart. Each stop sets a bar of intelligence measured from a reference model, so that "balanced" means the same thing next month even after every score has been rescaled. The fast end takes the cheapest rung within a few points of the reference; the smart end takes the smartest rung the plans allow and, where §6.4 applies, several models on one question.

One price per option. An option is a rung together with a way of feeding it the work: the whole thread, a brief written for the step, or a digest of a long result (§5). The router estimates each rung's quality for the task's kind of work: the general height where the map knows nothing more specific, the strength in that kind of work where it does, both corrected by the ledger. Among the options whose estimate clears the dial's bar and that pass the vetoes, it takes the one with the lowest price:

$$\text{price} = \text{money} \times \text{pace} + \lambda\, t_{\text{crit}} + p_{\text{fail}}\, c_{\text{rec}}$$

Money counts cached and uncached input from the cache ledger, plus the expected output. Pace is the quota multiplier. t_{crit} is the time the option adds to the critical path (§6), and λ is

what a second is worth for this task: high when someone is waiting, close to nothing for a job that runs overnight. The reader supplies it (§4). p_{fail} is the chance the call fails or is refused, from the provider's recent health and the ledger's record of refusals, and c_{rec} is what recovering would cost. These are routing prices: they decide which option wins, not what anyone pays.

Vetoes come first. Four checks run before any price. Privacy: content from a source the operator marked private goes only to approved services, and this is settled before anything leaves the machine. Policy: a table set by the operator says which vendors may take which classes of topic. Capacity: a task that does not fit a model's window, with room left for the answer, is ruled out for that model. Quota: a spent window rules its provider out until the reset.

4. Reading with Jev

A router is only as good as its reading of the situation. If it cannot tell a proof from a letter, or a step that needs thought from one that copies a value, every price in §3 is applied to the wrong thing.

What Jev is. Jev is a decision model from TypeSafe AI, released in September 2026 (TypeSafe AI, 2026). It reads text and returns typed answers: a choice among up to 255 named options, a score on an ordered scale, or the probability that a statement is true, each with its own confidence. It cannot write text, and one call can carry many questions. The vendor reports answers in 70 to 500 milliseconds; input costs a few cents per million tokens and output is free.

Three properties fit it to this job. Its answers are types, so the router computes with them directly and nothing in an answer can be read as an instruction. Its confidences are meant to be calibrated, so the router can weigh an answer by how sure it is. And it is quick and cheap enough to ask at every call: a read of a few thousand tokens costs a small fraction of the model call it routes.

Three reads. Thalamus asks Jev at three moments. The exact wording of each question is kept with the code; here is what each one asks and which options it offers.

- *The task read*, once per request and again whenever the plan changes: the kind of work (one of the map's domains); how hard it is, on five levels from a lookup to research; whether it touches a class of topic that some vendors refuse or answer with an official line; how urgent it is (someone waiting, needed today, whenever); and its shape (one answer, a list of independent parts, or a chain in which each part needs the last). The same call ranks the assistant's enhancements for the task (§7).
- *The step read*, before each model call that is not a plain continuation: what kind of step it is (plan, call a tool, read a result, write, check, answer); how much thought it needs (mechanical, routine or deep); how much of the conversation it needs (all of it, the recent part, or only the item in hand); how many mechanical steps probably follow; which pending items could run at the same time; and whether the step changes anything outside the conversation.
- *The outcome read*, after a call returns: done, worth a retry, stuck (the same failure twice), refused, or in need of a check.

Describe, then decide. Jev never names a model. It describes the situation, and the prices of §3 choose. The split keeps every decision explainable (these answers, these prices, this option), and it lets the choice follow a price change on the same day without touching the reader.

When the reader is unsure. Low confidence never makes a route cheaper or riskier. Below a floor the router takes the cautious option: no switch, no slice, the whole thread and the stronger of two rungs. If Jev does not answer at all the same holds, and every vendor with a topic restriction is left out, because nobody checked the topic.

Off the critical path. A read before every call would add up to half a second to each. So the step read for the next call is sent while the current tool runs, and a plain continuation reuses the last read. Most reads then cost no time at all.

One call, two jobs. The step read also carries the guard's questions from J11 (Serra, 2026a): whether the step matches what the user asked, and how hard it would be to undo. One call per step serves both papers, and both write to one ledger.

Privacy comes first. Jev is an outside service. Whether a task is private is decided before any read, from where the task came from: the operator marks sources, such as a private chat or a folder of company files, whose content may go only to approved services. For those tasks Jev reads only if the operator has approved it for that source. Otherwise a few local rules stand in for it, and the router takes the cautious options.

5. Where a model can change

A harness sees a task as a string of model calls, and at almost any point in that string it can give the next piece of work to a different model. What changes from one point to the next is what the new model receives and what is lost on the way (Table 1).

Table 1. Where a harness can hand work to a different model, from the whole request down to one call.

Switch point	The new model receives	Lost in the hand-over	Pays when
The request	the request	nothing	always: it is the first choice
Plan and work	the plan, or one unit of it	the planner's reasoning	planning and doing need different strengths
A unit of work	a brief written for the unit	the rest of the thread	the units are independent
The next step	the whole visible thread	warm cache, hidden reasoning, one voice	a long run of steps, a stuck step, or a cold cache
Reading a result	one long tool result	nothing the thread keeps	the result is long
The check	the work and what it claims	the builder's context, on purpose	the step commits to something or claims a fact
The finish	the content and the brief	the reasoning behind the content	the answer must read well in one voice or language
Inside a tool	whatever the tool sends	nothing	the tool's own job wants a specialist

Below the call sits the token: speculative decoding lets a small model draft tokens for a large one to check (Leviathan, Kalman and Matias, 2023). That happens inside one vendor's servers, out of a harness's reach, so the call is the finest switch point a router controls.

Most rows of the table cost nothing to switch at, because the new model starts on a fresh, short input anyway: a unit's brief, a single result, a piece of work to check, the content of a finished answer. One row is different. Handing the *next step* of a running conversation to another model hands over the whole thread, and the thread is where the money is.

5.1 The price of switching in the middle

Suppose the model in charge, the incumbent, has been working on a thread of C tokens. On its next step it re-reads the thread from its cache. A cheaper candidate, taking over, has to read the whole thread at its normal price the first time, and write it into its own cache to read it cheaply after that. Let the incumbent's input price be k times the candidate's, let a cached read cost r_b of the incumbent's price and r_s of the candidate's, and let the candidate's cache write cost w of its own. Over the next N steps, staying costs $N\,k\,r_b\,C$ and switching costs $w\,C + (N-1)\,r_s\,C$. Switching pays when

$$N > N^* = \frac{w - r_s}{k\,r_b - r_s}$$

and it never pays, on input alone, when $k\,r_b \le r_s$. The length of the thread cancels out: it sets how much is at stake, not when a switch starts to pay.

With the published multipliers (§3) the numbers are sobering (Figure 2). For a candidate five times cheaper, at the usual cache prices ($r = 0.1$, $w = 1.25$), N^* is about 2.9: the cheap model has to keep the thread for three steps before the switch pays. With the one-hour cache ($w = 2$) it is about 4.8. When the incumbent reads its cache for a twentieth of its price, as Opus 5.5 does, N^* rises to about 7.7 for the same five-fold gap, and a model ten times dearer that reads its cache for a fortieth, as Fable 5.1 does, lands at the same 7.7. For one step on its own the candidate need not write a cache at all and pays its normal price, so a single-step switch pays only when $k\,r_b > 1$: an incumbent more than ten times dearer at the usual cache price, twenty times at Opus 5.5's and forty times at Fable 5.1's. Coming back afterwards costs the incumbent a write of whatever was added in the meantime.

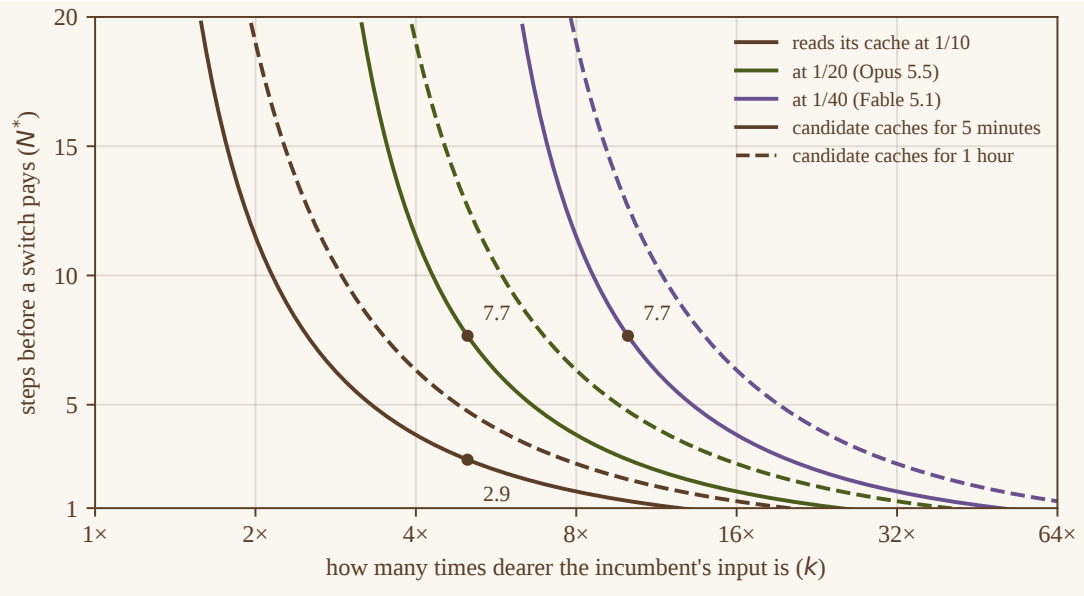


Figure 2. When handing the whole thread to a cheaper model starts to pay. The horizontal axis is how many times dearer the incumbent's input is; the vertical axis is the number of consecutive steps N^* the cheaper model must keep the thread before the switch pays. Each colour is a cache-read price for the incumbent (a tenth, a twentieth, a fortieth of its normal input); solid lines assume a five-minute cache write for the candidate, dashed lines a one-hour write. Computed from the published multipliers; below each curve a switch loses money.

Three things move the answer. Output is never cached, so a step that writes far more than it reads (a long file from a short brief) can pay for its switch at once, because its output price applies in full. A cache that has expired, after a tool that ran longer than the cache lives, leaves the incumbent reading cold too, and then the switch costs nothing extra. And the price ratio

k is set by the dial and the domain: the further the cheap model is from the incumbent, the sooner a long run pays.

5.2 Condense instead of carrying

The cheapest token is the one the thread never carries. A tool result of S tokens appended to the thread is written into the incumbent's cache once and re-read on each of the L steps that follow. A reader can take the result on its own, with a brief of what the thread is looking for, and return a digest of d tokens. With prices counted in units of the incumbent's input price, and the reader's price q, the digest pays when

$$(1 - d/S)\,(w + r_b\,L) > q$$

Because w is at least 1.25, any reader no dearer than the incumbent pays with a digest under a fifth of the result, even with no later steps at all, and the margin grows with every step that follows. This holds even when the reader *is* the incumbent's own model, which is the arithmetic behind sub-agents: the saving comes from not carrying the raw result, and a cheaper reader adds to it.

The risk is the detail that turns out to matter later. So the raw result stays on disk under a name, and the step read can ask for it back when the thread needs "the item in hand" in full. A digest that loses something costs one extra read, not a wrong answer.

5.3 What else is lost

Money is the part that can be counted. Three losses cannot be. Vendors return a model's reasoning signed, encrypted or summarised, and it cannot be handed to another vendor's model, so a switch passes on what the model wrote but not how it reasoned. Vendors format tool calls differently, so a switch across vendors needs a translation layer that has to be exact. And a reply stitched from several models reads unevenly, one paragraph in one voice and the next in another.

5.4 The switch policy

Five rules follow, and the router applies them without exceptions.

1. Choose freely at the fresh points. A new unit, a check, a finish and a digest all start without a warm thread to lose, so the router picks the best option there without any switching penalty.
2. Condense long results instead of carrying them (§5.2).
3. Switch a running thread only when it pays: for a run of mechanical steps longer than N^* (the step read estimates the run), for a stuck step, where a stronger model is cheaper than another failed try, or when the incumbent's cache has gone cold anyway.
4. Check with another family. The checker gets the work and what it claims, not the thread, and it comes from a different family from the builder's. A checker that shares the builder's model shares its blind spots.
5. Let one writer finish. The content can come from anywhere; the words the user reads come from one model, the one that writes that language and register best.

6. Finishing sooner

Models get faster every year. But an assistant's task is a chain, and a chain is as slow as its length: every call waits for the one before it and reads the conversation before its first token. So the scheduler's job is to make the chain shorter.

Where the time goes in one call is simple enough: waiting for the first token, which grows with uncached input; thinking; writing at the model's speed; then the tool. Across a task these add up along the longest chain of calls that depend on each other, the *critical path*. Amdahl's law (Amdahl, 1967) bounds the gain: if a fraction s of the work has to happen in order, no number of parallel workers finishes it in less than s of the time one worker takes. The design goes after that fraction first.

6.1 Plan as a graph

The planner writes the task as units, each with declared inputs, outputs and the files or records it changes. The scheduler starts every unit whose inputs are ready. Units that change the same thing never run at the same time; everything else may. Independence is checked from those declared changes, never assumed, because this is where parallel work goes wrong: two units that depend on each other, run side by side, each reason from an out-of-date view, and the result looks fine until the pieces meet. A request the task read calls a chain stays one unit.

6.2 Spend on the critical path, save off it

The graph gives every unit a slack: how much later it could finish without delaying the task. On the critical path time costs money, so those units get fast rungs and, when someone is waiting, a second copy if they run slow (§6.4). Off the path, slack is free time. The same units can take a cheaper, slower rung, or wait for a window that is about to refill. The price of §3 does this without a special rule, because λ charges only for time on the critical path.

6.3 Run things at once

- Many tool calls in one turn. Reads and searches that do not depend on each other go out together, which most models already support.
- Fan out on a shared start. Siblings that share a long start, such as a contract's definitions and the instructions, run on the same model, so the start is written into the cache once and every sibling reads it cheaply. Siblings with their own briefs go to whichever model is cheapest. The cache decides which kind of fan-out a task gets.
- Split long inputs. A document too long to read quickly is cut at natural boundaries, the pieces are read at the same time, and one model combines what the readers found.
- Pipeline. The next stage starts on partial output: the checker starts on the first file while the builder writes the second.
- Outline, then fill. A list-shaped answer gets its outline first and its parts written at the same time (Ning et al., 2023).

6.4 Guess ahead, and cover the tail

Prefetch. When the plan names what will be read next, a cheap reader condenses it while the strong model is still thinking. Work that turns out not to be needed is thrown away.

Hedge. When a call on the critical path runs past its usual time by a set margin, the scheduler sends a copy to another provider and keeps whichever answer arrives first (Dean and Barroso, 2013). A few calls are duplicated, and the long waits disappear.

Panels at the top of the dial. When the best model's lead over the best model from another vendor is small, several models answer the same question at once and the leader writes the final answer, seeing the others only at that point. This is parallel in time but multiplies the cost, and Self-MoA's finding keeps it off below the top stops of the dial (Li, W. et al., 2025).

6.5 More plans, more lanes

Every provider limits how many calls one account can run at once and how much a window holds. Running units on several providers at the same time adds lanes, and pace pricing spreads the work among them. The scheduler knows each provider's limit on concurrent calls, so a fan-out never queues behind itself.

6.6 When faster costs more

Running the same calls in a different order costs no extra tokens. Hedging, racing, panels and prefetching do, because they pay for work that may be thrown away. The model that collects a fan-out reads every part, so its input grows with the number of parts. The urgency in the task read decides the trade: a request with someone waiting buys speed, and an overnight job buys nothing but cheapness.

7. Which enhancement to reach for

An assistant carries more than models. Over time it collects *enhancements*: skills (a manual with scripts for one kind of task), recipes (a procedure written as steps) and plugins (code that adds tools). Each one holds the lessons of work done before, and none of them helps if the agent does not reach for it. Agents miss them often. They start from scratch on a task a recipe already covers, and they rarely see that something built for one job fits another: a loop written to check translations against their source fits any check of an output against a target, and a page built for sorting photos by hand fits any decision taken item by item. An agent usually chooses from names and one-line descriptions in its instructions, with nothing to rank them against the task in front of it.

7.1 A short list, in order

Jev already reads the task (§4). The same call carries one more question: which enhancement fits it. Each enhancement is an option, and each option carries a *card*, the text Jev reads about it. A choice question returns a probability for every option, so the answer is a ranking and not a single pick. Thalamus hands the agent the top of that ranking, in order and with the probabilities: the shortest run of entries that together hold 80 per cent of the probability, never more than six. One option always reads "none of these fits". When it leads, the list shrinks or is not shown, so the agent is never pushed toward a tool for its own sake.

The agent decides. Jev describes and ranks, but only the agent opens the files and sees the whole situation. It may take the first entry, one further down, one that is not on the list, or none. The list costs a few lines of the agent's context. It is refreshed when the plan changes and when a unit starts (§6.1), because a unit is often a different kind of task from the request that holds it.

A choice holds at most 255 options, and an assistant can collect more enhancements than that. So the enhancements are grouped into families (coding, writing, messages, documents, shopping and so on), and the call asks two things at once: which family the task belongs to, and, within each family, which enhancement fits. The probability of an enhancement is the product of the two answers. It is still one call, because a Jev call carries many questions. The cards cost little: a few hundred cards of sixty tokens each come to under twenty thousand input tokens, a small fraction of a cent at a few cents per million.

7.2 Made for it, or built like it

A card has two lines that matter. One says what the enhancement was made for. The other says how it works in general terms, without its subject: "compares an output with a target and repeats until they match", or "shows a person one item at a time and records a decision on each". The first line finds the tasks it was written for. The second finds the others, the tasks that need the same shape of work in another subject. Over time a card gains a third part, the kinds of task it has actually served that its author never had in mind (§7.4).

For the first few entries, a second small read says how each one fits: made for this task, fits by its structure, or covers part of it. It goes out while the first model call runs, so it costs no time, and it tells the agent whether to follow an entry as written or adapt it. When the top two entries each cover part of the task, the list says they fit together. If the ledger sees the same two used together for the same kind of task three times, Thalamus proposes a merged enhancement, drafted from the two and from the transcripts that combined them, for the owner or the recipe author to accept (Serra, 2026b). A kind of task done three times with no enhancement at all gets the same proposal, drafted from the transcripts that did it by hand.

7.3 Watching what the agent uses

The label is what the agent did, as the harness saw it, not what the agent says it did. An enhancement counts as used when the agent opens its manual or procedure or calls one of its tools. For every task the ledger (§8) records the list that was shown, in order and with its probabilities; what was used, on the list or off it; how it was used, as written, adapted or merged; and how the task ended: done, retried, or corrected by the owner.

That label is biased, and the bias would teach the loop to agree with itself. An agent tends to take the first entry because it is first. A loop that treats every pick as the truth rewards Jev for whatever it already put on top, and the ranking freezes. Search engines have the same problem, since people click the top result because it is on top, and their answer carries over (Joachims, Swaminathan and Schnabel, 2017). Each pick is weighted by how surprising it is. If the agent takes the entry at rank r in a share $\pi(r)$ of all tasks, whatever the list says, a pick at rank r counts $1/\pi(r)$. With $\pi(1) = 0.7$ a first-place pick counts about 1.4, and with $\pi(4) = 0.05$ a fourth-place pick counts 20. The rates are measured the way search engines measure them: now and then, on overnight jobs where it costs little, the list is shown in a shuffled order.

Two more rules keep the labels honest. A pick that led to a failed task or a retry is no evidence that it fitted. And the owner's correction, "you have a skill for this", is the strongest label there is. It is kept as a pinned case that no later change may lose.

7.4 A better question every night

Jev has no weights of ours. What Thalamus can change is the text Jev reads: the instruction of the enhancement question and the cards. So the text is what learns. Every night:

1. The ledger yields the misses: tasks whose used enhancement ranked low or was absent, weighted as in §7.3, and confident lists whose top entry went unused.

2. For each group of similar misses, a writer model (itself a routed call) proposes a small edit: a line added to the card that should have ranked higher, a sharper structure line, or a narrower card for an enhancement that drew probability it did not earn.
3. Each edit is replayed. Jev reads a held-out set of past tasks with the old cards and with the new ones, and the edit is kept only when it ranks the used enhancements higher on that set (the mean reciprocal rank, weighted as in §7.3) and every pinned case keeps its rank or improves.
4. Kept edits are small, and every version of every card is stored, so one step back is always possible.

This is prompt optimisation from errors: a model proposes new wording from the cases that went wrong, and a held-out score decides (Pryzant et al., 2023; Khattab et al., 2023). The same run checks calibration. When Jev gives the used enhancement 0.6, it should turn out to be used about six times in ten, and a mapping learned from the ledger corrects the probabilities shown to the agent, so that "0.6" keeps its meaning while the cards change.

Replay sends past task reads to Jev again, so it follows the privacy rule of §4. Only tasks from sources approved for Jev enter the replay set. For tasks from private sources the list comes from local word matching against the cards, which is weaker, and those tasks never leave the machine.

8. The design

Put together, Thalamus is one loop with a ledger underneath (Figure 1). A request goes through it like this.

1. Intake. The source of the request sets its privacy before anything leaves the machine.
2. Task read. Jev reads the task: kind of work, difficulty, topic class, urgency and shape. The same call ranks the enhancements, and the agent gets the short list (§7).
3. Plan. A request shaped as one answer or a chain stays one unit. A request shaped as independent parts goes to a planner, itself a routed call, which returns the graph of §6.1.
4. Schedule. Every unit whose inputs are ready starts, within each provider's concurrency limit.
5. Each call. The step read (sent while the previous tool ran) describes the step. The router lists the options, a rung with a way of feeding it; drops what the vetoes forbid; prices the rest (§3); applies the switch policy (§5.4); and makes the call with a ladder behind it.
6. The ladder. If a call fails, the reason picks the next rung: a rate limit moves to another provider, a task too long to a larger window, a refusal to another family, a timeout to a faster rung. A model the user picked by hand gets no ladder; an explicit choice is never answered by a different model.
7. Outcome read. Done, retry, escalate, or send to a check.
8. Check and finish. Work that commits to something or claims a fact is checked by another family (§5.4); the words the user reads come from one writer.
9. Ledger. Every decision is written down with the reader's answers and their confidences, the options with their prices, the vetoes applied, the pick, the model that answered in the end, the cost, the time and the outcome, with the short list shown and the enhancement used.

The loop fits in a few lines:

```
on request(task):
    privacy   = from_source(task)           # before anything leaves
```

```
    read      = jev.task_read(task, privacy, cards)  # also ranks enhancements
    shortlist = top(read.enhancements, share=0.8, most=6)
    offer(shortlist)                          # the agent picks
    graph     = plan(task, read) if read.shape == "parts" else [unit(task)]
    run every ready unit of graph at once, within provider limits

run(unit):
    until done:
        step    = jev.step_read(unit)       # sent while the last tool ran
        options = [(rung, feed) for rung in map
                   for feed in (thread, brief, digest)]
        options = [o for o in options if vetoes_pass(o, privacy, read)
                                     and quality(o, read) >= bar(dial)]
        pick    = min(options, key=price)   # money x pace + time + risk
        pick    = switch_policy(pick, step, cache_ledger)
        result  = call(pick, ladder=by_reason(options))
        ledger.write(step, options, pick, result)
        outcome = jev.outcome_read(result)
    check(unit) if commits_or_claims(unit)
    ledger.write(shortlist, used_enhancement(unit), outcome)

every night:
    picks = ledger.enhancement_picks(weight = 1 / pi(rank))
    for edit in writer.propose(cards, misses(picks)):
        if replay(held_out, cards + edit) > replay(held_out, cards) \
           and pinned_cases_hold(cards + edit):
            cards.apply(edit)               # every version kept
```

Learning without a training run. Each night the ledger updates the router's estimates. For every kind of work, kind of step and rung it counts successes, retries, refusals, cost and time. An estimate starts at the public map and moves toward the ledger's own record as the evidence grows: a rung with ten outcomes stays close to the map, and one with a thousand follows its record. So that the record does not freeze on early winners, the router now and then tries an option it would not normally take, only on overnight jobs where a worse answer costs little. A vendor that refuses legitimate work in a class of topic twice within a month is skipped for that class. The kinds of work with no public benchmark get their first real evidence this way. The enhancement cards learn on the same night, as §7.4 describes.

What it asks of the harness. The router needs to see every model call before it is made: the model, the input and the cache counts that come back. The scheduler needs units with declared inputs, outputs and changes. The checker needs a second vendor. The short list needs a place in the agent's context and a way to see which enhancement the agent opens or calls. Everything else is data: the daily map, the quota windows, the cache ledger, the enhancement cards and the ledger of outcomes.

9. The afternoon, routed

Back to the four requests of §1, through the design.

Table 2. The four requests of the introduction, each read by Jev and routed.

Request	What the reader says	Route
Proof in group theory	mathematics, research-grade, a chain, someone waiting	the strongest measured mathematician at high effort on the whole thread; a model from another family checks the proof; no fan-out, because each step needs the last
Letter in Swahili	languages, routine, one answer	the model with the best measured coverage of Swahili, at low effort, in one call
Log of a break-in	security, attack code, from a private source	privacy settles it first: only approved, self-hosted models; the log is split by host and read at once; the strong approved model reads the digests
600,000-token contract	long documents, routine per clause, parts, someone waiting	split, read at once by cheap long-context readers on two providers, combined by a strong model; doubtful clauses checked by a third family

The contract shows most of the design at once (Figure 3). The task read finds long documents, routine work, a list-shaped answer and someone waiting. The planner makes the contract's definitions and the instructions a shared start, and cuts the body at clause boundaries into twelve pieces of 50,000 tokens. Twelve readers run at the same time on a cheap long-context model, six on each of two providers, and each reads the shared start from its provider's cache after the first reader has written it there. One reader runs slow and is hedged on the other provider. A strong model combines the readers' lists: it reads twelve short lists of candidate clauses, not the contract. The doubtful clauses go to a checker from a third family, with the clause and the definitions. The answer comes from the strong model, which now holds the whole picture in a few thousand tokens.

The short list works on the same afternoon. For the contract it leads with a procedure for searching a long document in pieces, made for the job. In second place is a loop written to check translations against their source, which fits the check of doubtful clauses by its structure: compare a piece of text with a reference and repeat until they agree. The agent takes both. That night the loop's card gains a line saying it also checks clauses against definitions, and the next contract finds it one place higher. For the Swahili letter "none of these fits" leads, the list is not shown, and nobody is pushed toward a tool.

10. What would prove it

This is a design. These tests would show whether it works, and each one could fail.

1. The reader on real requests. Label a few hundred real requests and their steps by hand, with personal details removed before anything is sent out, and compare Jev's task and step reads with the labels, including whether its confidence matches how often it is right.
2. Outcomes against honest baselines. Run the same tasks through Thalamus, through the single best model at the same cost, through a router that chooses once per request, and through a fixed fallback list. Compare success, cost and wall time. The strongest rival explanation is that one good model with an ordinary fallback list gives most of the benefit; this test has to rule it out.

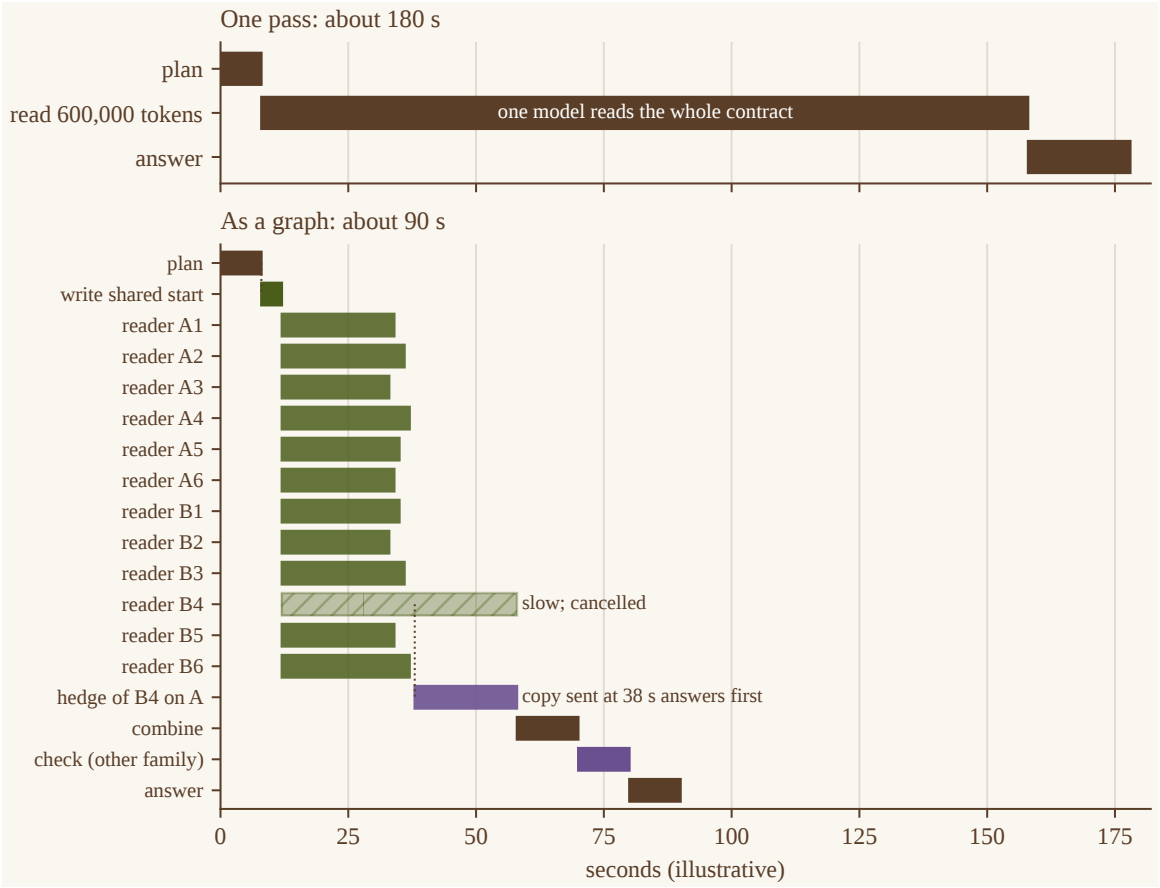


Figure 3. The contract search, read in one pass and as a graph. Top: a single long-context model plans, reads 600,000 tokens and answers, one call after another. Bottom: the planner writes a shared start; twelve readers on two providers read at the same time; one slow reader is hedged on the other provider; a strong model combines, a checker from another family looks at the doubtful clauses, and the strong model answers. Durations are illustrative, chosen to show the shape, not measured.

3. The cache ledger. Compare the cached and uncached tokens the ledger predicts for each call with what the provider bills. The switch rule is only as good as those numbers.
4. Digests. On tasks whose answer depends on one detail inside a long result, compare digests with the raw result, and count the details lost and the extra reads needed to recover them.
5. Speed. Compare the measured wall time with the critical-path estimate, and the quality of parallel runs with serial runs of the same tasks. A parallel plan that is faster and worse is a failure.
6. Quota. Record how often a window runs dry and how much intelligence stays available through the week, with pace pricing on and off.
7. Overhead. Measure the time and money the reads and the router add to each call, with the reads sent ahead and without.
8. The short list. On a few hundred tasks labelled by hand with the enhancement that should have been used, measure how often it is on the list and at which rank, against a list built from word matching alone. Repeat after a month of nightly revisions, on tasks the revisions never saw, and measure with shuffled lists how much the agent follows position. If a month of revisions does not beat the first cards on unseen tasks, the loop is learning to agree with itself.

11. Limits

Jev belongs to a third party. It is an early-access service, its speed and accuracy figures are the vendor's own, and its price may be subsidised. When it is down the router falls back to the cautious options, which are the expensive ones. Typed questions see only what they ask about, so a situation none of them names is routed as if it were ordinary.

The map borrows public benchmarks, which may not match any one person's work, and several kinds of work have no benchmark at all until the ledger fills in. Plan prices are estimates. The policy table is the operator's choice, not a measurement of what vendors do.

A digest can lose the detail that matters, and the recovery read of §5.2 helps only if the thread notices. A switch across vendors loses the hidden reasoning, and only outcomes can show what that costs. Parallel work on units that really depend on each other produces confident, wrong results, and the declared changes of §6.1 are only as good as the planner that declares them. Every decision point is also a place to be wrong, which is why the ledger records all of them.

The short list learns from what the agent does, and an agent follows lists. Position weights, outcomes and the owner's corrections guard against a loop that agrees with itself, but only the corrections are clean labels, and they are rare. The cards are text written by a model, and a bad edit can narrow an enhancement without anyone noticing. The replay and the stored versions guard against that, and they are only as good as the held-out set.

12. Conclusion

Thalamus sends each call, not only each request, to the model that fits it. Jev reads the task and each step in typed answers; the router prices every option in money, quota, cache and time at one dial's setting; a scheduler runs everything that can run at once and spends only on the critical path; and a ledger turns every decision into evidence for the next. Jev also ranks the assistant's own enhancements, and every night the text it reads gets a little better at predicting which one the agent reaches for. The arithmetic of caching sets the style of the whole design: fresh points are free, whole threads are expensive, so the router moves slices of work instead of conversations.

The vision is the secretary of the first page. She does not do every job herself and she does not hand the whole file to whoever is free. She gives each piece to the one who does it best, gives them only the pages they need, hands each the procedure that fits, even one written for another job, and has three people working while the fourth is still reading. If the tests of §10 hold, the daily map of the model market stops being something a person reads and becomes something the assistant uses at every call.

References

Amdahl, G. M. (1967). *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.* AFIPS Spring Joint Computer Conference, 483–485.

Anthropic (2026). *Prompt caching.* Claude Platform documentation. https://platform.claude.com/docs/en/build-with-claude/prompt-caching (read 30 September 2026).

Artificial Analysis (2026). *Artificial Analysis Intelligence Index.* https://artificialanalysis.ai/

Chen, L., Zaharia, M., and Zou, J. (2023). *FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance.* arXiv:2305.05176. https://arxiv.org/abs/2305.05176

Dean, J., and Barroso, L. A. (2013). *The Tail at Scale.* Communications of the ACM, 56(2), 74–80.

Epoch AI (2026). *Benchmarking hub.* https://epoch.ai/benchmarks

Hugging Face (2026). *Security incident disclosure — July 2026.* 16 July 2026. https://huggingface.co/blog/security-incident-july-2026

Joachims, T., Swaminathan, A., and Schnabel, T. (2017). *Unbiased Learning-to-Rank with Biased Feedback.* WSDM 2017. arXiv:1608.04468. https://arxiv.org/abs/1608.04468

Khattab, O., Singhvi, A., Maheshwari, P., et al. (2023). *DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines.* arXiv:2310.03714. https://arxiv.org/abs/2310.03714

Leviathan, Y., Kalman, M., and Matias, Y. (2023). *Fast Inference from Transformers via Speculative Decoding.* arXiv:2211.17192. https://arxiv.org/abs/2211.17192

Li, W., Lin, Y., Xia, M., et al. (2025). *Rethinking Mixture-of-Agents: Is Mixing Different Large Language Models Beneficial?* arXiv:2502.00674. https://arxiv.org/abs/2502.00674

LiteLLM (2026). *Routing*; *Reliability: fallbacks, retries and cooldowns*; *[Beta] Auto Routing.* https://docs.litellm.ai/docs/routing; https://docs.litellm.ai/docs/proxy/reliability; https://docs.litellm.ai/docs/proxy/auto_routing

Ning, X., Lin, Z., Zhou, Z., Yang, H., and Wang, Y. (2023). *Skeleton-of-Thought: Prompting LLMs for Efficient Parallel Generation.* arXiv:2307.15337. https://arxiv.org/abs/2307.15337

OckBench (2025). *OckBench: Measuring the Efficiency of LLM Reasoning.* arXiv:2511.05722. https://arxiv.org/abs/2511.05722

Ong, I., Almahairi, A., Wu, V., et al. (2024). *RouteLLM: Learning to Route LLMs with Preference Data.* arXiv:2406.18665. https://arxiv.org/abs/2406.18665

Panda, P., Magazine, R., Devaguptapu, C., et al. (2025). *Adaptive LLM Routing under Budget Constraints.* arXiv:2508.21141. https://arxiv.org/abs/2508.21141

Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. (2023). *Gorilla: Large Language Model Connected with Massive APIs.* arXiv:2305.15334. https://arxiv.org/abs/2305.15334

Pryzant, R., Iter, D., Li, J., Lee, Y. T., Zhu, C., and Zeng, M. (2023). *Automatic Prompt Optimization with "Gradient Descent" and Beam Search.* arXiv:2305.03495. https://arxiv.org/abs/2305.03495

Qin, Y., Liang, S., Ye, Y., et al. (2023). *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs.* arXiv:2307.16789. https://arxiv.org/abs/2307.16789

Serra, O. (2026a). *AMYGDALA — A Digital Amygdala for AI Agents: The Little Voice Agents Are Missing.* J-series J11, v6.7.

Serra, O. (2026b). *PREFRONTAL: The Recipe Execution Substrate — How a Composable, Self-Authoring, Looping Workflow Engine Becomes an Agent's Executive Function.* J-series J13, v4.8.

Tang, Y., Cetin, E., Xu, J., et al. (2026). *Sakana Fugu Technical Report.* arXiv:2606.21228. https://arxiv.org/abs/2606.21228

TypeSafe AI (2026). *Introducing System One models and Jev.* https://typesafe.ai/blog/introducing-system-one-models-and-jev

Wang, J., Wang, J., Athiwaratkun, B., Zhang, C., and Zou, J. (2024). *Mixture-of-Agents Enhances Large Language Model Capabilities.* arXiv:2406.04692. https://arxiv.org/abs/2406.04692

J-series paper J19 | Version 4.1 | 30 September 2026