

When Should a Reasoning Model Stop? Budgeting Test-Time Compute Under Deadlines

Veera Ravindra Divi
Independent Researcher
Austin, TX
ravi3divi@gmail.com

Abstract

A reasoning service can improve benchmark accuracy by spending more inference compute and still become less useful when users share a constrained accelerator pool. Extra branches consume cache, verification consumes time, and long generations delay unrelated requests. This white paper develops a deployment decision protocol for allocating reasoning compute under deadlines, monetary budgets and tenant constraints. It synthesizes test-time scaling and inference-serving research, including Certainindex, SART and probabilistic demand modeling. The proposed contribution is an operational contract connecting admission, continuation, stopping and accounting to correctness delivered before a deadline. A worked example illustrates the accounting; a reproducible evaluation protocol specifies the evidence needed to validate the design. No new model, implemented scheduler, benchmark release or measured performance improvement is claimed.

The service boundary changes the objective

Consider a shared assistant that receives a short factual request while another tenant launches several long mathematical reasoning branches. The second request might become more accurate with additional inference, yet its branches occupy memory and decoding capacity while the first request waits. Evaluating each request independently hides that externality. Production usefulness depends on the joint distribution of answer quality, completion time and resource consumption across all admitted and rejected requests.

Test-time computation has become a practical design choice. Snell et al. investigate difficulty-conditioned allocation between reasoning strategies; s1 exposes reasoning duration through budget forcing; DeepSeek-R1 demonstrates an important training route for reasoning behavior [1, 2, 3]. These works justify treating inference effort as controllable. They do not establish that more tokens always improve correctness, that every task benefits, or that benchmark gains survive shared-service congestion.

The question addressed here is therefore specific: how should a service decide whether a request deserves another reasoning increment when that increment competes with other requests and a finite deadline? The intended audience is an ML platform team controlling both application policy and enough serving telemetry to measure its effects. For opaque external APIs, portions of the design become observational because the team cannot inspect hidden reasoning, cache allocation or internal scheduling.

The recommended deployment unit is a reasoning program: the complete set of generations, retrievals, tool calls and verification steps needed to deliver one answer. Optimizing only individual calls can improve token throughput while leaving program completion unchanged. This paper proposes a service contract and an evaluation method, not a claim of first invention for adaptive reasoning or deadline scheduling.

What existing evidence establishes

The literature separates into three complementary layers. At the reasoning layer, Snell et al. show that allocation effectiveness depends on question difficulty and the mechanism used to spend compute [1]. s1 demonstrates explicit intervention in thinking duration [2]. DeepSeek-R1 studies reasoning through reinforcement learning and distillation [3]. These results motivate task-conditioned budgets; they do not provide calibrated probability estimates for every production request.

At the program layer, Certainindex measures answer stabilization and integrates early exit and dynamic allocation into Dynasor [4]. SART combines branch sampling, early stopping and pruning to address long reasoning paths and concurrent memory pressure [5]. Hermes uses probabilistic demand graphs for scheduling multi-stage applications and prewarming backends [6]. These are essential prior art. A proposal that merely renames confidence-based stopping or program-aware scheduling would add little.

At the runtime layer, PagedAttention addresses dynamic KV-cache allocation and sharing, SGLang exploits structured multi-call programs and prefix reuse, DistServe separates prefill from decoding, and Sarathi-Serve uses chunked prefill to manage throughput and latency [7, 8, 9, 10]. These mechanisms affect the cost of a reasoning decision. A token is consequently an accounting unit, not a fixed amount of time, memory or money across hardware, models and context lengths.

The remaining engineering problem is composition: deciding what evidence is sufficient to grant another reasoning increment while preserving an explicit user-facing service objective. The protocol below connects these layers through deadlines, execution budgets and conservative stopping. Its value must be assessed against Certainindex, SART and Hermes where their implementations and assumptions permit comparison, rather than against an artificially weak fixed-token policy alone.

Define the contract before selecting a policy

For request i , record arrival a_i , absolute deadline d_i , monetary ceiling b_i , tenant class t_i and an acceptance predicate A_i . The predicate might be exact mathematical correctness, passing held-out tests, or a human-adjudicated requirement. Choose the predicate before evaluating policies. A request is useful only when it satisfies A_i and is delivered by d_i . Tool-dependent applications must also satisfy their policy and authorization constraints.

Define $U_i = 1 \{ A_i \text{ is satisfied and } c_i \leq d_i \}$, where c_i is completion time. Useful throughput over an observation window T is $(\sum_i U_i) / T$. Report this alongside offered-load success, admitted-load success, latency distributions, total expense and rejection rate. A policy can inflate admitted success by rejecting difficult requests, so admitted-only metrics cannot serve as the headline. If correctness cannot be observed online, use delayed audit labels for evaluation and state that the controller operates with uncertain proxies.

Separate four quantities that are often conflated: a maximum generation length, a total reasoning-program compute budget, an end-to-end wall-clock deadline and a tenant entitlement. A program that samples eight short branches can exceed the cost of one long branch. A prefill-heavy request can consume substantial resources before generating a token. A cheap request can still miss a deadline behind other work. Each constraint needs its own ledger.

Reserve enough budget for producing the final answer and any required verifier. An early-stop decision that leaves an unfinished chain without an answer is a service failure. Admission should distinguish hard constraints from soft targets: enforce billing ceilings and authorization boundaries directly, but describe latency as a probabilistic service target unless the runtime genuinely offers a hard real-time guarantee.

A four-decision control loop

Admission comes first. Estimate a conservative completion-time distribution for the smallest supported answer path, including queue delay, prefill, generation, verification and finalization. Admit only when that path is plausibly feasible under the service's chosen risk tolerance and available budget. Otherwise return an explicit rejection, offer an asynchronous class if the product supports it, or produce a permitted bounded fallback. Count each outcome in offered-load reporting.

Continuation is incremental. At a checkpoint, the controller chooses among extending one branch, creating an additional branch, verifying existing candidates, returning a candidate, or abstaining. Let $\widehat{\Delta q}(i, k)$ denote an estimated improvement in acceptance probability from action k , and $\widehat{\Delta c}(i, k)$ its expected additional cost. A practical ordering signal is $\widehat{\Delta q} / \widehat{\Delta c}$, subject to deadline feasibility, memory limits and tenant entitlement. This ratio is a heuristic requiring validation; it is not a proof of global optimality for coupled queues. Include verifier and controller overhead in $\widehat{\Delta c}$. Require predicted marginal value to clear a configured threshold after uncertainty adjustment; otherwise select the feasible return or abstention path.

Stopping should combine at least three observations: whether a candidate meets the verifier's acceptance rule, whether independent evidence suggests further reasoning has low marginal value, and whether sufficient slack remains. Agreement among samples is useful evidence but can reflect shared errors. Certainindex provides prior art for stabilization-based signals [4]. A controller must not relabel stability as certainty or assume a self-evaluation score is calibrated without held-out testing.

Execution accounting closes the loop. Charge all generated and discarded branches, verifier calls, repeated prefill, external tool operations and finalization work to the originating program where measurable. Keep measurements separate from estimates. If the API does not expose hidden reasoning compute, state that limitation and report the observed bill or provider usage fields rather than inventing an internal FLOP estimate.

Reconsider decisions at bounded intervals and after major events, such as a candidate completing or a tool returning. Excessively frequent checks add overhead; excessively coarse checks waste compute after the answer is already adequate. Cancellation must propagate to queued and running work. If a backend cannot cancel immediately, account for residual execution and measure cancellation delay before promising savings.

Figure 1 shows the controller's admission and checkpoint decisions. Every continuation reserves resources before dispatch, and completion includes verification and finalization rather than stopping at the end of a reasoning trace.

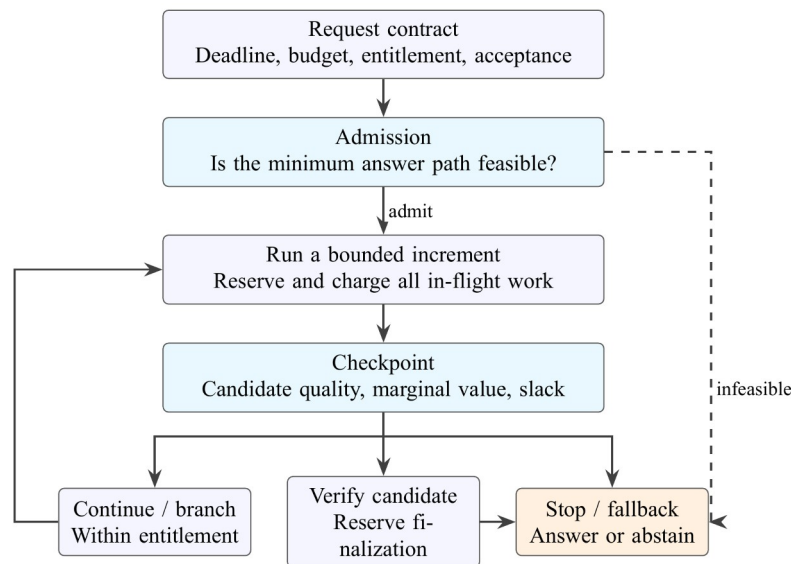


Figure 1. Reasoning control loop. Additional reasoning, branching, and verification compete for the same deadline and program budget. Stopping returns an accepted candidate or a declared fallback; agreement alone is not proof of correctness.

Table 1. Reasoning control decisions and the costs they must include.

Decision	Required inputs	Failure to avoid
Admit	Minimum path, queue, deadline	Accepting infeasible work
Continue	Marginal value, remaining budget	Spending without service benefit
Verify	Candidate, verifier cost	Forgetting finalization
Stop	Acceptance evidence, slack	Mistaking agreement for truth
Account	All branches and residual work	Claiming unrealized cancellation savings

Shared resources, fairness and failure handling

A single-request budget becomes a shared-service policy only when it respects cache occupancy and scheduling interactions. Prefix sharing can make a new branch cheaper than its full context suggests, while divergent continuations still require additional cache blocks [7, 8]. Record physical occupancy or a defensible proxy; do not simply multiply token count by a constant and call the result measured memory. Longer-lived contexts also change the opportunity cost of retaining speculative branches.

Separate the logical controller from the serving scheduler. The controller decides which work remains valuable; the runtime decides which feasible work runs next. DistServe and Sarathi-Serve offer different ways to reduce phase interference and manage batching [9, 10]. They do not automatically implement task-level utility optimization. When combining mechanisms, measure the integration overhead and retain their original latency definitions so time-to-first-token is not confused with final-answer completion.

Tenant fairness needs explicit policy. A greedy marginal-value rule can repeatedly favor easy requests or tenants with cheaper models. Give each tenant a configured reservation or weighted share, then allocate spare capacity opportunistically. Report deadline success and expenditure separately by tenant and difficulty stratum. Fairness is a product and contractual choice; no single weighted objective establishes that every tenant has received equitable treatment.

Protect the controller against bad estimates. Sudden workload changes, verifier failures and cache eviction can invalidate optimistic predictions. Use bounded increments, conservative slack and a maximum cumulative budget as backstops. Log the estimate available at each decision rather than reconstructing it after seeing the outcome. If the verifier is unavailable, follow a declared fallback, such as abstention or a lower-assurance answer class. Do not silently treat an absent check as a pass.

Treat retries as part of the same program. A transport timeout does not create a fresh entitlement, and duplicate tool effects require independent idempotency controls. Per-request compute limits alone cannot prevent a tenant from flooding the service with many requests; admission-rate controls remain necessary. These operational boundaries keep reasoning optimization from becoming a path around ordinary capacity management.

Illustrative accounting example

The following numbers are hypothetical arithmetic, not measurements or recommended production thresholds. A request arrives with a six-second deadline. Its minimum path is estimated to require one second of queueing, half a second of prefill, two seconds of reasoning, half a second of verification and half a second of finalization. The planned total is 4.5 seconds, leaving 1.5 seconds of slack. These components are estimates with uncertainty, not guaranteed durations.

At the first checkpoint, suppose another reasoning increment needs an estimated 0.8 seconds and verification then needs another 0.5 seconds. The combined 1.3 seconds fits the nominal slack but leaves only 0.2 seconds for prediction error. A conservative policy could decline the increment when its configured uncertainty reserve exceeds that margin. A policy based only on remaining token allowance might approve it and miss the deadline despite staying within budget.

Now suppose three branches each consume one unit of compute and the verifier consumes another half-unit. The program has used 3.5 units even if only one branch appears in the answer. Returning early after two completed branches does not save the third unit if the backend finishes it before cancellation takes effect. Charge the actual completed work; separately report requested cancellation and realized avoided work. This distinction is necessary for credible cost claims.

Finally, assume the same action delays another tenant's short request past its deadline. The first request's local quality gain cannot alone determine whether the action benefits the service. An evaluation must include both requests and the offered workload. The example illustrates why reasoning budgets, service deadlines and shared-capacity decisions need one consistent accounting boundary. It supplies no evidence about the accuracy or latency of a real implementation.

Evaluation that could validate the design

Begin with two workload families: verifiable reasoning tasks, where correctness can be scored reproducibly, and multi-call applications that expose variable tool delays and program structure. Keep training, controller calibration and final evaluation disjoint. Publish task identifiers, model versions, prompts, decoding settings, maximum context, hardware, runtime revisions and acceptance rules. If public benchmark tasks may have entered model training, state contamination uncertainty instead of treating the split as a complete remedy.

Use open-loop arrivals that are independent of completion times. Include steady traffic, bursts, mixed lengths and tenant imbalance. A closed-loop generator that waits for responses can reduce offered load when the service slows down and mask overload. Sweep offered load through saturation, report all rejections and unfinished requests, and drain or explicitly censor the observation window. Include tool-time variability and cold-start behavior when those are part of the intended deployment.

Compare at least a fixed-budget policy, a task-conditioned static policy, a stabilization-based adaptive policy, and relevant program-aware systems [4, 5, 6]. Equalize model weights, total resource entitlement, verifier access and tuning effort as far as technically possible. When integration differences prevent a fair direct comparison, label the

comparison as a separate configuration study. Do not multiply speedups from unrelated papers or use their headline numbers as an aggregate baseline.

Measure useful throughput, offered-load acceptance before deadline, accuracy without deadline censoring, completion latency, total compute or billed cost, cache occupancy, canceled-work waste and tenant-stratified outcomes. Include uncertainty across repeated trials and independently sampled arrival traces. Report confidence intervals for policy differences using a task- or trace-aware procedure; individual tokens are not independent experimental samples. Retain task-level outcomes so averages can be audited.

Ablate the marginal-value estimator, uncertainty reserve, cancellation support, tenant reservation and verifier budget. Sweep calibration error and inaccurate output-length predictions. Test whether the policy still helps under distribution shift rather than only under its training workload. The deployment gate should require a predeclared service improvement with bounded regressions in critical tenant classes. The exact threshold depends on the application; this white paper does not supply a universal numeric gate.

A deployable decision record

Make each continuation decision inspectable without requiring a transcript of internal reasoning. A decision record should contain the program identifier, arrival time, absolute deadline, budget remaining, reserved finalization cost, current candidate status, selected action, estimator version and predicted incremental cost. Log the information available at decision time. Append observed duration, charged resources and outcome later. This separation prevents retrospective explanations from being mistaken for the controller's actual evidence.

For an initial implementation, use a small finite action set and a held-out lookup table conditioned on workload class, observed progress and remaining slack. This is easier to audit than an unconstrained learned controller. Each table cell needs a sample count and an uncertainty estimate; sparse or shifted cells should fall back to the fixed policy. Update the table only through a versioned calibration process. The finite action set is a proposed engineering starting point, not an established optimal design.

Run counterfactual shadow decisions cautiously. A shadow policy can record that it would have stopped, but it cannot observe whether an unexecuted continuation would have improved the answer. Offline traces also reflect the behavior policy that generated them. Avoid reporting counterfactual savings as measured service gains. A bounded randomized experiment, where ethically and operationally appropriate, can supply more direct evidence while still enforcing the same hard budgets and deadline safeguards.

Declare how final-answer quality is graded when a controller returns a shorter explanation. Mathematical answer correctness, test execution and human usefulness are different outcomes. Evaluate them separately where relevant. An earlier answer can satisfy an exact-answer checker while omitting a requested explanation; conversely, an attractive explanation can conceal an incorrect result. The acceptance predicate must reflect the actual service promise, including any required uncertainty statement or abstention behavior.

Limits and implementation sequence

The strongest limitation is that the proposed controller has not been implemented or evaluated here. Its architecture is an evidence-backed synthesis and a testable design, not a demonstrated improvement over existing schedulers. Marginal correctness gains may be difficult to estimate, and self-consistency can confidently reinforce a wrong answer. Verifiable mathematics and code tasks provide cleaner feedback than open-ended advisory work, so results from one family should not be generalized without further evidence.

Deadline prediction also depends on the serving stack. A black-box API can expose billing and completion time while hiding batching, cache pressure and reasoning tokens. In that setting, enforce the observable budget, use measured response-time distributions and avoid claims about internal resource efficiency. Preemption, fine-grained cancellation and physical memory accounting require runtime support; SART explicitly identifies preemptible scheduling as additional work [5].

A practical sequence starts with end-to-end accounting and a fixed policy, adds held-out calibration and shadow decisions, then enables bounded continuation decisions for a limited workload class. Expand only after the evaluation protocol exposes the relevant regressions. Preserve a rollback path to the fixed policy and retain decision traces without storing unnecessary sensitive prompts. The resulting system should make one promise well: spend additional reasoning compute only when the available evidence justifies it within the declared service contract.

Acknowledgment

Prepared as an independent research white paper. The views expressed do not represent an employer. Proposed mechanisms require implementation and empirical validation.

References

- [1] C. Snell, J. Lee, K. Xu, and A. Kumar, "Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters," arXiv:2408.03314v1, Aug. 2024. <https://arxiv.org/abs/2408.03314v1>
- [2] N. Muennighoff et al., "s1: Simple test-time scaling," arXiv:2501.19393v3, Mar. 2025. <https://arxiv.org/abs/2501.19393v3>
- [3] DeepSeek-AI et al., "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," arXiv:2501.12948v1, Jan. 2025. <https://arxiv.org/abs/2501.12948v1>
- [4] Y. Fu et al., "Efficiently Scaling LLM Reasoning with Certainindex," arXiv:2412.20993v2, May 2025. <https://arxiv.org/abs/2412.20993v2>
- [5] Y. Wang, Y. Jiang, B. Cui, and F. Fu, "Thinking Short and Right Over Thinking Long: Serving LLM Reasoning Efficiently and Accurately," arXiv:2505.13326v1, May 2025. <https://arxiv.org/abs/2505.13326v1>
- [6] Y. Liu et al., "Efficient Serving of LLM Applications with Probabilistic Demand Modeling," arXiv:2506.14851v1, Jun. 2025. <https://arxiv.org/abs/2506.14851v1>
- [7] W. Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention," arXiv:2309.06180v1, Sep. 2023. <https://arxiv.org/abs/2309.06180v1>
- [8] L. Zheng et al., "SGLang: Efficient Execution of Structured Language Model Programs," arXiv:2312.07104v2, Jun. 2024. <https://arxiv.org/abs/2312.07104v2>
- [9] Y. Zhong et al., "DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving," arXiv:2401.09670v3, Jun. 2024. <https://arxiv.org/abs/2401.09670v3>
- [10] A. Agrawal et al., "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve," arXiv:2403.02310v3, Jun. 2024. <https://arxiv.org/abs/2403.02310v3>