Patient Bytes: A Reliability-Budgeted Scheduler for Agentic LLM Workflows

Hannah B. Pasandi
UC Berkeley
Berkeley, USA
h.pasandi@berkeley.edu

Negar Arabzadeh
UC Berkeley
Berkeley, USA
negara@berkeley.edu

Melissa Pan
UC Berkeley
Berkeley, USA
melissapan@berkeley.edu

Tianyin Xu
UC Berkeley, CA, USA
University of Illinois, IL, USA
tyxu@berkeley.edu

Sina Darabi
Barcelona Supercomputing Center
Barcelona, Spain
sinad1367@gmail.com

Mohammad Hosseini
Shahid Beheshti University
Tehran, Iran
m-hosseini@sbu.ac.ir

Hamed Haddadi
Imperial College London and Brave Software
London, UK
h.haddadi@imperial.ac.uk

Abstract

Production agentic LLM workflows are asynchronous. The human waits on the answer, not on the next token, so per-request latency is the wrong axis to optimize. The right objective is reliability per workflow. We define the *reliability budget*, the bytes a workflow spends on speculation, verification, retry, and judging; on our software-engineering workload it is close to half of workflow bytes. Gateways spend this budget by accident today because schedulers act on the call while correctness lives in the workflow. Patient is an agent-gateway scheduler that spends an application-declared byte budget explicitly through three rules, a budget guard, dependency priority, and load-aware routing. The rules fit in a gateway plug-in, the budget itself comes from a twenty-run profiling recipe rather than tuning, and we prove a $(1+\varepsilon)$ bound on reliable completion against an offline oracle. On SWE-bench, Patient lifts reliable completion from 0.63 to 0.91, seventeen points over the strongest baseline, at completion times within 1% of the default. The emulator, the five policies, and every script behind the artifact findings are available at github.com/hanabhp/patient-bytes.

CCS Concepts

• **Software and its engineering** → **Operating systems**; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks*; • **Computing methodologies** → *Intelligent agents*.

Keywords

LLM serving, agentic workflows, scheduling, reliability, gateway

1 Introduction

A software-engineering agent picks up a GitHub issue. A generator call drafts a patch, a tool call runs the tests, a verifier call checks the draft; on disagreement the agent retries and a judge call breaks the tie. Every call transits the gateway, the reverse proxy, standardized by MCP, through which an organization's agents reach their model and tool backends, and every call arrives there as a separate request. Nothing tells the gateway these requests belong to one workflow, so it serves each on arrival under the chat rule of returning the next token fast. But no human is waiting on that token. The human is waiting on the workflow, which returns minutes after it began.

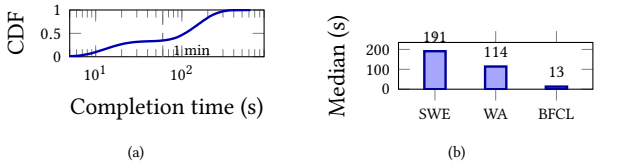


Figure 1: Completion times under the default scheduler, shown as a pooled CDF (a) and per-benchmark medians (b); dashed line at one minute.

Not every agent runs this exact loop; the shape is the production norm. Pan et al. surveyed 306 practitioners and ran twenty case studies of deployed agents [19]. In their data, LLM-as-a-judge verification is the second most common evaluation method, reported for 52% of deployed agents behind human review at 74%; 66% of deployments accept end-to-end response times of a minute or longer, and 17% set no latency limit at all. Agent traffic does not fit the latency goal mainstream ML-systems work optimizes.

As agents move from prototypes to always-on services, the gateway that admits, orders, and routes their calls becomes the resource controller for the workload, the place an agentic OS decides what runs next. Classical OS and serving schedulers inherit a latency objective from processes a user waits on and requests bounded by a tail-latency SLO; agent workloads fit neither. The unit that matters is the workflow, and the property that matters is correct completion, not how fast the next call returns. We call this gap the *unit mismatch*. Agentic AI raised the unit of computation from the call to the workflow, and the scheduler's objective must rise with it [18]. Patient is our mechanism for closing the gap, at the gateway the agent OS already routes every call through.

Existing serving systems optimize the latency axis. vLLM [10] maximizes throughput under a latency target, DistServe [46] and Sarathi-Serve [1] optimize p99 inference latency, and Niyama [4] co-schedules requests by latency SLO. Cortex [17] makes the workflow visible to the serving layer through stage isolation, and Patient composes with a Cortex or Niyama backend beneath its gateway. Each is right for its workload and misapplied on the agentic traffic above them.

For agentic traffic the right objective is reliability per workflow. Figure 1 shows why first-token latency no longer fits. It plots completion times for 12,000 workflows from SWE-bench, WebArena, and BFCL under the default scheduler. Almost all finish far above the sub-second window chat serving targets. At that timescale, shaving the next token barely moves the workflow; what matters is whether it completes correctly. The application enforces correctness with verification round-trips, retries on disagreement, and model-as-a-judge calls; the bytes a workflow spends on this traffic are its reliability budget, and they are a large share of the total (§2).

The gateway spends this budget by accident today, serving verifier and retry calls like any other traffic, without knowing what they are for. This paper makes the spend deliberate, trading longer per-request latency for a higher reliable completion rate. Patient accepts two request headers per workflow, an identifier and a byte budget the application will spend on being correct, and applies three rules at every scheduling decision, a budget guard, a dependency priority that lifts blocking verifiers, and load-aware routing that respects model-version pins. The rules fit in a gateway plug-in of roughly 700 lines of Go, and we prove a $(1+\varepsilon)$ bound on reliable completion against an offline oracle that knows each workflow's verification needs (§4). We make three contributions.

• **Characterization.** We characterize the asynchronous agentic latency profile across three emulated benchmarks and quantify the reliability budget: verification, retry, judge, and speculation traffic reaching half of workflow bytes and, service being byte-proportional, of service time (§2, §6).

• **Design.** We design Patient, a reliability-budgeted gateway scheduler with a $(1+\varepsilon)$ bound, and a profiling recipe that sets the budget from twenty pilot runs, not tuning (§3, §4).

• **Evaluation.** On one seeded workload, Patient lifts reliable completion over four latency-oriented baselines; the pilot recipe matches exact per-workflow budgets, and dependency priority starves no workflow class (§6).

Broader impact. The same two headers can carry other workflow-level objectives such as a cost ceiling, an energy budget, or a fairness share across tenants. Reliability becomes a quantity an application declares and a platform provisions for, the way latency SLOs work today, not an accident of retry loops buried in agent code. Because the mechanism is a gateway plug-in, adoption is a deployment decision, not a platform rewrite.

2 The Reliability Budget

We define the *reliability budget* of an agentic workflow as the maximum bytes the gateway will spend on speculation, verification, retry, and judging before the workflow completes or is declared failed.

Bytes are what the gateway can meter without parsing provider-specific payloads, what the backend bills, and what the network moves; for text models they track tokens closely. Calls are not comparable; a single verifier over a long context can cost more than ten short generator calls, so a budget counted in calls would license expensive verification while starving cheap generation. And because a backend serves bytes at a rate, a byte budget doubles as a cost budget on the provider invoice and a time budget on the clock (§4). The budget of a workflow is the sum of the byte costs of its stages, whatever mix of calls produces them. It has four components.

Verification bytes. A second model called explicitly to score the first model's output. On the wire it is a request-response pair with the previous output as input; because the verifier rereads what it checks, a verified stage costs roughly twice the generation it verifies (§3).

Retry bytes. The same request issued again after the verifier disagrees. This is an application-layer reissue, distinct from TCP retransmission, to the same model version (intra-version) or a different one (cross-version).

Judge bytes. A third model called to break generator-verifier ties or to assign the confidence scores that gate human handoff. The pattern appears in five of Pan et al.'s twenty case studies, all high-stakes [19].

Speculation bytes. Bytes a speculative generator spends exploring an alternative the workflow may later discard. Speculation counts against the budget like the other three components; unlike them it never gates the workflow, which is why the scheduler of §4 ranks it last. Not every stage gates the workflow. A verifier whose disagreement triggers a retry is *blocking*. Until it returns, the runtime cannot decide whether the next generator stage is needed, so a budget exhausted before that verifier runs strands the workflow undecided. A speculative generator is *non-blocking*; its bytes count, but delaying it costs nothing. The scheduler spends the budget on blocking stages first, and the dependency edges E tell the gateway which is which.

Why existing schedulers mis-budget. Default LLM routing treats every request as independent. Join-shortest-queue balances on backend depth, blind to whether a request is the first attempt or the third retry. Niyama's QoS prioritization [4] ranks by per-request deadline slack, blind to which requests share a workflow and which verify which. Smoothness controllers such as SODA [2] treat retries as failures to suppress rather than as the application enforcing reliability. The result, quantified in §6, is a budget spent without coordination, with verifier orderings that strand the next generator stage behind budget exhaustion. Patient spends the budget explicitly.

3 Emulation Setup

We evaluate Patient in a discrete-event emulation of the gateway and its backends (Figure 2). The model schedules a workflow's generator, verifier, retry, judge, and speculation stages through the gateway against a pool of LLM backends. Each stage carries a byte cost and a service time proportional to it, so a stage's share of bytes and its share of service time coincide by design; offered load adds queueing delay and shrinks the effective per-workflow

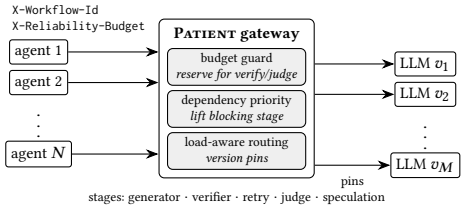


Figure 2: Emulated gateway and backends. N agents issue workflow stages through the Patient gateway to M LLM backends; our runs use N=8 and M=3.

budget through shared capacity. The five schedulers differ only in how they order ready stages, whether they reserve budget for a pending verifier, and whether they issue retries.

Three benchmarks drive the workload. SWE-bench Lite [7] asks the agent to read a repository, localize a bug, request a patch, and verify it, four to six LLM calls per task. WebArena [47] hands the agent a navigable web environment, intermediate in call count. BFCL, the Berkeley function-calling benchmark [22, 23], is single-call. Each benchmark is a workload profile whose stage counts, verification and disagreement rates, and per-stage byte costs are calibrated to Pan et al.'s production survey [19] and to published plan-length and token-waste measurements [13, 26]; we draw 4,000 workflows per benchmark per scheduler. The resulting SWE-bench reliability share, close to half of workflow bytes, sits just under the 59.4% of tokens Tokenomics measures in the iterative review stage of multi-agent software engineering [26], so the calibration is, if anything, conservative.

Setting the budget. Budgets come from profiling, not tuning. For each benchmark we draw twenty pilot workflows from the same profile and set B to the 75th percentile of their realized reliability bytes. Finding 8 shows this recipe within noise of exact per-workflow budgets and robust to using five pilots instead of twenty, so the one number the application must declare is a profiling output.

Emulation Rationale. Three reasons. First, a paired comparison needs the identical, seeded workload under all five schedulers, and live model endpoints do not replay. Second, provider rate limits and nondeterministic backend latencies confound scheduler effects with serving noise. Third, the sweeps of §6 amount to more than 60,000 workflow executions, infeasible against metered endpoints. Nothing in Patient requires emulation; the plug-in targets a live MCP gateway, and a live deployment is the immediate next step (§8). The harness is seeded, every parameter is documented in the workload generator, which is the honest place to disagree with us, and the emulator, policies, tests, and scripts behind the artifact findings and every figure accompany the paper.

4 The Reliability-Budgeted Scheduler

Patient is an agent-gateway scheduler that accepts a reliability budget from the application and spends it explicitly. It runs above the transport and below the agent framework; it knows nothing about the contents of the calls and everything about the network state.

Setting. We model a workflow as a directed acyclic graph $W = (V, E, B)$, with V the stages, E the dependency edges, and B the declared budget in bytes. A stage is a generator, verifier, judge, or speculation call. The gateway never sees the full graph in advance. The application reveals W one stage at a time, with edges exposed as prior stages complete, so Patient schedules an online, incrementally revealed graph and needs no plan up front. A schedule σ assigns stages to backends over time, respecting E and not exceeding B; the workflow is *reliably completed* if every required generator stage has a verifier-agreeing output before the budget runs out.

What the application supplies. Exactly two request headers, `X-Workflow-Id`, which ties calls to their workflow, and `X-Reliability-Budget`, the bytes it will spend on being correct. The dependency edges arrive with the stages themselves, emitted by the agent runtime as it sequences calls, not by developer annotation, and escalation to human review on budget exhaustion follows the application's existing failure contract [19] rather than a new input. Nothing else is asked of the developer.

Objective. Over valid schedules Π, maximize the reliable completion rate $R(\sigma) = \Pr[W \text{ reliably completed under } \sigma]$ subject to scheduled bytes not exceeding B. Chat-serving first-token objectives and vLLM's tail-latency SLO [10] are the degenerate corner where $B \to \infty$ and verifier stages are absent.

Algorithm. At each decision, Patient applies three rules in order. We first tried only Rules 1 and 3; the verifier-priority rule was added after early runs showed SWE-bench workflows sometimes ran the verifier last, leaving work that was correct but reported incomplete because verification arrived too late to gate the next attempt.

- **Rule 1. budget guard.** If bytes scheduled so far plus the candidate stage would exceed B, decline it. The workflow may complete unreliably or escalate to human review per the application's contract.
- **Rule 2. dependency priority.** Among schedulable stages, prefer those that close a blocking dependency in E; verifiers and retries that gate further work rank ahead of generators that block nothing, and speculation ranks last. Ties age, a ready stage's priority improves with waiting, so no stage class waits unboundedly (Finding 9).
- **Rule 3. load-aware backend selection.** Route the chosen stage to the shortest backend queue, subject to the model-version pin a retry carries.

Does Patient optimize time at all? Only through work conservation. Rule 3 keeps backends busy and balanced, and within the budget every admitted stage runs as early as its dependencies allow. The application caps how long reliability may take through the budget itself, since at a given serving rate a byte budget is a time budget, and Table 1 shows the completion-time cost of the reordering stays within 1% of the latency-first baselines. A workflow a user would rather do by hand is one whose application set B too high; the budget makes that choice explicit.

Theorem 4.1. *Assume (i) the budget B is within a constant factor of the offline-oracle budget $B^\star$, (ii) per-stage costs are known within ε of their realized values, and (iii) the dependency graph is acyclic. Then Patient achieves reliable completion rate $R(\textsc{Patient}) \geq (1+\varepsilon)^{-1} R(\textsc{Oracle})$.*

In words. The oracle knows, before scheduling, which stages each workflow will need verified and what each will cost; Patient learns both online. Assumption (i) says the declared budget lands in the regime Figure 4b maps and the recipe of §3 targets. Assumption (ii)

bounds cost-estimation error; header-declared or history-estimated stage sizes satisfy it in practice. Assumption (iii) holds for agent plans, which do not wait on themselves. *Proof idea.* Patient processes stages greedily in topological order, and Rule 2 guarantees no schedulable verifier is starved. An exchange argument does the rest; moving any verifier later in Patient's order, in favor of a stage that blocks nothing, cannot raise the number of workflows verified within budget, so Patient's order matches the oracle's up to the cost-estimation slack, which the ε absorbs. The result is a competitive ratio under these assumptions, not global optimality, and when the budget is mis-sized Patient degrades to the budget-guard baseline rather than losing the guarantee silently (Finding 7).

The three rules on one workflow. Return to the coding agent of §1, where a generator drafts a patch, a tool stage runs the tests, a verifier checks the result, and, on disagreement, a retry and a judge follow, all under budget B. Under Default, each stage is scheduled by arrival and queue depth, so the verifier and judge compete with unrelated chat traffic and can land after the budget guard has declined the retry, leaving the workflow correct internally but reported incomplete. Under Patient, Rule 2 lifts the verifier because it closes the dependency the retry needs, Rule 1 reserves enough of B for that verifier and the retry, and Rule 3 keeps the retry on the draft's model version so the verifier's judgment still applies. The workflow completes reliably inside B, later in wall clock than a latency-first schedule would return the individual calls, which is the tradeoff the application asked for.

5 Implementation

We realize the three rules as a plug-in for an MCP gateway, roughly 700 lines of Go that intercept the routing decision and emit a per-decision record of which rule fired and the workflow's remaining budget. The plug-in exists; the numbers in §6 do not come from it. They come from the emulation of §3, which is why we claim no kernel work and no live-deployment result. The application contract is the two headers of §4, no new transport.

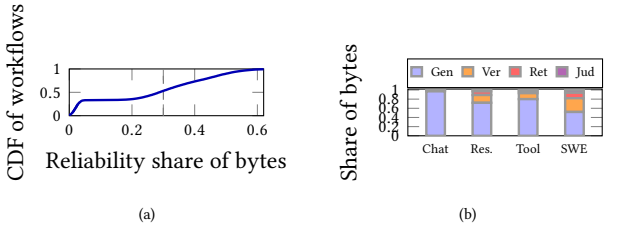
6 Evaluation

Setup. We compare five schedulers through the same gateway. *Default* is the unmodified gateway router. *JSQ* is join-shortest-queue. *SODA-A* is our agent-stage adaptation of SODA, the smoothness-optimized adaptive-bitrate controller deployed in production video streaming [2]; each workflow stage is treated as a chunk and admission is smoothed. We include it as the closest production controller that paces admission rather than racing it; its objective, smoothness for a watching viewer, is precisely what agent traffic lacks, and it sheds the verification traffic Patient protects. The name collides with an unrelated stream-processing scheduler for System S [36]; SODA-A derives from the streaming controller only. *Niyama-Sarathi* adapts Niyama's QoS prioritization [4] to the gateway routing decision over a Sarathi-Serve backend model [1], the strongest published competitor. We measure *reliable completion*, the share of workflows whose outputs a verifier confirms before the budget runs out. Each paragraph reports one finding.

Finding #1. Completion takes minutes, not milliseconds, and the spread across benchmarks is 15×. Under Default, median completion is 191 s on SWE-bench, 114 s on WebArena, and 13 s on BFCL (Table 1), and the pooled distribution in Figure 1 puts the median above one minute. The share of workflows completing inside one second, the chat-LLM QoE target, is negligible; first-token latency and workflow completion are different quantities.

Table 1: Scheduler comparison. RCR is reliable completion rate, p50 and p95 are completion times in seconds, and Verif% is the verifier byte share, equal to its time share (§3).

Bench.	Sched.	RCR	p50	p95	Verif%
SWE-bench	Default	0.633	191	327	30
	JSQ	0.617	193	324	30
	SODA-A	0.396	180	292	30
	Niyama-Sarathi	0.740	196	329	33
	Patient	**0.909**	192	324	35
WebArena	Default	0.640	114	206	28
	JSQ	0.653	114	204	27
	SODA-A	0.498	106	188	28
	Niyama-Sarathi	0.728	115	208	31
	Patient	**0.845**	116	206	32
BFCL	Default	0.896	13	29	2
	JSQ	0.982	13	29	2
	SODA-A	0.905	13	28	2
	Niyama-Sarathi	0.873	13	28	2
	Patient	**0.983**	13	29	2



(a) (b)

Figure 3: Reliability share of workflow bytes as a CDF (a; dashed line at 0.30) and its composition by agent type, Chat, Research, Tool-use, and SWE (b).

Finding #2. Reliability traffic is a first-order share of workflow bytes, and equally of service time. We classify each call into productive bytes (generator) and reliability bytes (verifier, retry, judge, speculation). Figure 3a plots the distribution of the reliability share across workflows as a CDF; nearly half of workflows spend over 30% of their bytes on reliability traffic, and the step near zero is BFCL, whose single-call tasks have almost nothing to verify. Because service time is proportional to bytes in the model (§3), the same curve gives the distribution of reliability *time* share; the fraction of execution time the scheduler can reorder is not small. In the released reference emulator, reliability operations occupy 32.5 to 32.6% of scheduled service time under all five schedulers at its documented operating point of 24 concurrent workflows on four backends at offered load 1.5, so the scheduler changes *when* reliability work runs, not how much there is.

Finding 3. The reliability share grows with agent autonomy. The mix of verification, retry, and judge traffic shifts with autonomy (Figure 3b). Chat workflows spend almost everything on generation; software-engineering workflows spend close to half on verification, retry, and judging combined. A scheduler that prices the budget

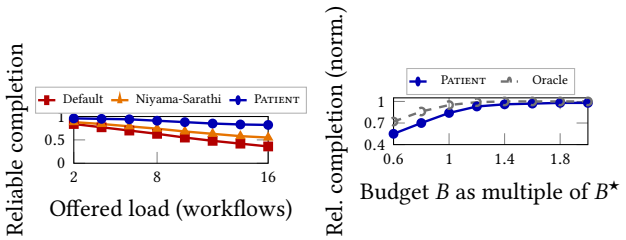


Figure 4: Reliable completion on SWE-bench against offered load (a) and against budget size relative to the oracle $B^\star$ (b).

has to see this composition, because a retry that closes a blocking verifier is worth more than a generator call that closes nothing.

Finding #4. Patient lifts reliable completion without a latency bill. Patient improves reliable completion over all baselines on SWE-bench and WebArena (Table 1), by 17 and 12 points over Niyama-Sarathi and by 28 and 21 points over Default. The gain is largest on SWE-bench, the most verification-heavy workload, where budget-aware ordering closes blocking dependencies earlier. BFCL, single-call with minimal verification, shows the smallest gap, the predicted degenerate corner. Per-workflow p50 and p95 stay within a small factor of the latency-optimizing baselines, the tradeoff Pan's data says applications accept. Verif% rises under Patient because the budget admits verification a latency-first schedule reaches too late; because byte share equals time share (§3), that spend is four to five points of service time on the two verification-heavy workloads, and it buys the completion gain.

Finding #5. The gain survives load. Reliable completion degrades with offered load for every scheduler on the SWE-bench workload; Patient stays above the baselines across the range (Figure 4a). Scheduling is global, with every ready stage of every workflow competing in one queue; Niyama-Sarathi is the closest competitor. Default falls fastest because it does not see the completion target; JSQ and SODA-A track it closely and are omitted for legibility. Whether dependency-heavy jobs can starve light ones is Finding 9.

Finding #6. Each rule earns its place. Removing Rule 1 (budget guard) degrades reliable completion under tight budgets to the Default level. Removing Rule 2 (dependency priority) raises p95 wall-clock time without changing median completion, confirming verification ordering as the source of the improvement. Removing Rule 3 leaves correctness intact but degrades p95 under bursty arrivals.

Finding #7. The bound is visible, and a mis-sized budget degrades gracefully. Figure 4b sweeps budget sizes on the SWE-bench workload. As B reaches $B^\star$, reliable completion approaches the oracle; below $B^\star$ it degrades toward the budget-guard baseline rather than collapsing, and above $B^\star$ extra budget buys little. A budget near $B^\star$ is the operating point, and the application's reliability target chooses it.

Finding #8. A pilot recipe sets a near-tight budget. Table 2 evaluates the recipe of §3 in the released reference emulator. The p75-of-twenty-pilots budget matches exact per-workflow budgets at 99.2% reliable completion; p50 gives up 2.1 points, p90 buys nothing, and five pilots do as well as twenty. Assigning a budget in advance, the assumption behind Theorem 4.1(i), is a cheap profiling step, not a tuning burden.

Table 2: Pilot budget recipes against exact per-workflow budgets in the reference emulator, ten seeds (Finding 8).

Budget recipe	B (KB)	RCR (%)	Time (s)
p50 of 20 pilots	79.7	97.1 ± 3.4	1.77
p60 of 20 pilots	88.2	98.3 ± 2.2	1.75
p75 of 20 pilots	102.4	**99.2 ± 1.8**	1.76
p90 of 20 pilots	122.1	99.2 ± 1.8	1.76
p75 of 5 pilots	99.6	98.8 ± 2.0	1.76
Exact per-workflow		99.2 ± 1.8	1.76

Finding #9. Dependency priority does not starve dependency-light workflows. Rule 2 is global, so a natural worry is that verification-heavy workflows crowd out light ones. We mix twelve heavy workflows (6 to 12 stages) with twelve light ones (1 to 2 stages) in the reference emulator, ten seeds at the same operating point. The light class does better under Patient than under any baseline on every metric, with 90.0% reliable completion and a 1.80 s p95 against 50.8% and 2.72 s under Default and 74.2% and 1.90 s under JSQ, while the heavy class reaches 100%. Lifting blocking verifiers drains queues sooner, and the aging term in Rule 2 bounds how long any ready stage waits, so global prioritization redistributes waiting rather than concentrating it.

7 Related Work

Serving-layer schedulers. Niyama [4] is the closest contemporary system, with hybrid SRPF-EDF prioritization over QoS classes. Niyama maximizes requests meeting a latency SLO where Patient maximizes workflows completing with verifier-agreed correctness inside a byte budget, and the two compose; a Niyama backend under a Patient gateway is the strongest baseline in §6. Orca [42], vLLM [10], DistServe [46], Llumnix [29], and Splitwise [21] operate inside the serving stack on per-request objectives. Cortex [17] is closest in spirit, exposing the workflow to the serving layer through stage isolation where Patient exposes it to the gateway through a budget. Tetris [27] offloads KV cache for agentic workloads beneath the gateway, and PCAPS [11] brings precedence awareness to data-processing graphs.

DAG and microservice schedulers. Dependency-aware scheduling long predates agents; Condor's DAGMan orders jobs by DAG precedence [3], and Airflow ships the same model as workflow infrastructure [30]. Patient keeps the precedence idea and changes the objective and the currency; the graph is revealed online, the scarce resource is reliability bytes rather than slots, and the guarantee is a completion bound against a budget-aware oracle. Microservice schedulers spend latency budgets across the stages of a call graph; GrandSLAm apportions per-stage slack from an end-to-end SLA [9], and Erms profiles stage latencies to schedule shared services under SLA guarantees [16]. Patient makes the same move one layer up with a different currency, bytes for correctness traffic rather than milliseconds for a response, and with the verifier-generator distinction those systems do not need.

Budgeted agentic execution. Concurrent work prices agent execution at the plan level. On Time, Within Budget allocates models and parallel samples to maximize completion probability under joint

budget and deadline limits [33]; Yang and Zhu derive closed-form token allocations trading latency, reliability, and cost [40]; Sherlock speculates across workflow stages for reliability [25]. These act in the planner or the runtime and choose *what* to run; Patient acts at the gateway and chooses *when and where* what the application already chose gets served, so they compose. Workflow-aware serving engines, Agentix [15], HexAGenT [24], Helium [31], and Continuum [12], optimize workflow latency or cache reuse below the gateway, and a policy-runtime survey maps the space [43]. None budgets reliability traffic, which is the axis Patient adds.

Agent frameworks, benchmarks, and observability. Patient schedules workflows produced by runtimes such as ReAct [41], AutoGen [37], and Reflexion [28]; we change only the gateway beneath them. Unified Agentic Interfaces argues for system-boundary observability [6], EnergAt attributes energy at thread granularity [5], and AgentSight traces agents with eBPF [45]; Patient lifts attribution, and its scheduling consequence, to the workflow.

OS mechanisms for agents. AgentCgroup measures tool-call-driven resource bursts inside the agent sandbox and enforces limits in-kernel with eBPF [44]; Patient makes the same move at the gateway, where the scarce resource is reliability bytes rather than memory. DMI separates an agent's planning policy from its interaction mechanism at the OS interface [34]; the budget header is the same split at the serving boundary, the application declares intent and the gateway owns the mechanism. Branch contexts give agents fork, explore, and commit operations for parallel solution paths [32]; exploration of that kind multiplies the speculation and retry traffic a budget prices.

Agent cost and verification. The budget prices traffic that recent work measures at the agent layer. Tokenomics finds iterative review takes most tokens in multi-agent software engineering [26], reading and rereading dominate coding-agent trajectories [35], and trajectory reduction and context management remove large fractions of the spend [8, 38]. These treat the spend as inefficiency to remove; Patient treats verification spend as productive and budgets it.

Production and the raised unit. Pan et al.'s production study documents the asynchronous, verification-heavy profile that motivates Patient [19]; their outlook argues verification, not capability, is the bottleneck for useful agents [18]. Parallel efforts build systems support for agent exploration [39] and agent-first data systems [14].

8 Discussion

Where Patient fits. Workloads with a reliability target, nontrivial verification traffic, and a human who waits on the workflow rather than the next token. Chat and voice agents have none of these, and Patient should not be deployed for them; the BFCL corner of §6 is where its advantage shrinks to nothing. The scope that matters is the asynchronous, verification-heavy middle and tail of the production distribution, where the bytes and dollars concentrate.

Why a budget is the right abstraction. Verification, retry, and judge calls are already on the wire; teams size them implicitly through retry counts and client timeouts, and the cost surfaces only on the invoice. A reliability budget moves the decision to the layer that can act on it, one number per workflow that turns an implicit cost into a schedulable one, the accounting done where the plan, not the call, names the spend. The same opacity appears inside a call, where profiles of reasoning models collapse a hidden chain-of-thought trace and the visible answer into one figure a scheduler cannot separate [20].

Why this belongs at the gateway of the agent OS. The gateway is the userspace control point through which an organization's agents already reach every model and tool backend. Patient lives there, not in the kernel CPU scheduler; the three things it needs, a budget contract, a propagated workflow identifier, and a per-decision log, are state the gateway already holds. Whether the gateway counts as inside the OS is a boundary question; the OS-level decision for this workload, what runs next, is made there, on state it already has, with no new hardware or transport.

Threats to validity. Our numbers come from a discrete-event emulation calibrated to published production measurements, not a production fleet; absolute gains will move with the workload mix and backend. The qualitative results are more robust than the point numbers; the reliability budget is a measurable fraction of bytes and of service time, the latency and completion axes are uncorrelated, and prioritizing blocking verifiers improves reliable completion at a latency cost the application accepts. A mis-sized budget is the main failure mode and it degrades to the budget-guard baseline rather than breaking (Findings 7 and 8). A live gateway deployment would test what the emulation abstracts, queueing against real backends, provider rate limits, real retry timing, and byte-time separation once service is no longer byte-proportional.

Limitations. Patient assumes the application can name a reliability target and that per-stage costs are estimable within ε; workloads with unbounded or unknowable verification needs fall outside the guarantee. The version pins of Rule 3 are the only place Patient touches multi-version coexistence; pricing cross-version verification fully is future work.

9 Conclusion

Agentic LLM workflows are asynchronous, and the network has been optimizing the wrong axis. We defined the reliability budget, close to half of workflow bytes and, in our model, of service time; designed Patient to spend it with a $(1+\varepsilon)$ bound; showed a twenty-run pilot recipe matches exact per-workflow budgets; and lifted SWE-bench reliable completion from 0.63 to 0.91, seventeen points over the strongest baseline, within 1% of Default's completion times.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, 117–134.

[2] Tianyu Chen, Yiheng Lin, Nicolas Christianson, Zahaib Akhtar, Sharath Dharmaji, Mohammad Hajiesmaili, Adam Wierman, and Ramesh K. Sitaraman. 2024. SODA: An Adaptive Bitrate Controller for Consistent High-Quality Video Streaming. In *Proceedings of the ACM SIGCOMM 2024 Conference*. Association for Computing Machinery, 613–644. doi:10.1145/3651890.3672260

[3] Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. 2007. Workflow Management in Condor. In *Workflows for e-Science*. Springer, London, 357–375. doi:10.1007/978-1-84628-757-2_22

[4] Kanishk Goel, Jayashree Mohan, Nipun Kwatra, Ravi Shreyas Anupindi, and Ramachandran Ramjee. 2025. Niyama: Breaking the Silos of LLM Inference Serving. *arXiv preprint arXiv:2503.22562* (2025). doi:10.48550/arXiv.2503.22562

[5] Hongyu Hè, Michal Friedman, and Theodoros Rekatsinas. 2023. EnergAt: Fine-Grained Energy Attribution for Multi-Tenancy. In *2nd Workshop on Sustainable*

Computer Systems (HotCarbon '23). Association for Computing Machinery. doi:10.1145/3604930.3605716

[6] Yanpeng Hu and Yusheng Zheng. 2025. Unified Agentic Interfaces is All You Need for AI Agent Observability. Preprint.

[7] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *The Twelfth International Conference on Learning Representations (ICLR)*.

[8] Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A. Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. 2025. ACON: Optimizing Context Compression for Long-horizon LLM Agents. *arXiv preprint arXiv:2510.00615* (2025).

[9] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth EuroSys Conference (EuroSys '19)*. Association for Computing Machinery, 1–16. doi:10.1145/3302424.3303958

[10] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, 611–626. doi:10.1145/3600006.3613165

[11] Adam Lechowicz, Rohan Shenoy, Noman Bashir, Mohammad Hajiesmaili, Adam Wierman, and Christina Delimitrou. 2025. Carbon- and Precedence-Aware Scheduling for Data Processing Clusters. In *Proceedings of the ACM SIGCOMM 2025 Conference*. Association for Computing Machinery, 1241–1244. doi:10.1145/3718958.3750478

[12] Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. 2025. Continuum: Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live. *arXiv preprint arXiv:2511.02230* (2025).

[13] Keyu Li, Junhao Shi, Yang Xiao, Mohan Jiang, Jie Sun, Yunze Wu, Dayuan Fu, Shijie Xia, Xiaojie Cai, Tianze Xu, Weiye Si, Wenjie Li, Dequan Wang, and Pengfei Liu. 2026. AgencyBench: Benchmarking the Frontiers of Autonomous Agents in 1M-Token Real-World Contexts. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[14] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, Matei Zaharia, Alvin Cheung, Natacha Crooks, Joseph E. Gonzalez, and Aditya G. Parameswaran. 2025. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. Preprint.

[15] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. 2026. Agentix: An Efficient Serving Engine for LLM Agents as General Programs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 2443–2459.

[16] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 2023. Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS 2023)*. Association for Computing Machinery, 62–77. doi:10.1145/3567955.3567964

[17] Nikos Pagonas, Yeounoh Chung, Kostis Kaffes, and Arvind Krishnamurthy. 2025. Cortex: Workflow-Aware Resource Pooling and Scheduling for Agentic Serving. Preprint.

[18] Melissa Pan, Yuxuan Zhu, Jared Quincy Davis, Riccardo Cogo, Lakshya A. Agrawal, Negar Arabzadeh, Xiaoyuan Liu, Huanzhi Mao, Sid Pallerla, Tianneng Shi, Alexander Xiong, Alessandro Basile, Emmanuele Lacavalla, Shuyi Yang, Diana Arroyo, Paul Castro, Daniel Kang, Joseph Gonzalez, Koushik Sen, Dawn Song, Ion Stoica, Matei Zaharia, and Marquita Ellis. 2025. Useful Agentic AI: A Systems Outlook. Preprint.

[19] Melissa Z. Pan, Negar Arabzadeh, Riccardo Cogo, Yuxuan Zhu, Alexander Xiong, Lakshya A. Agrawal, Huanzhi Mao, Emma Shen, Sid Pallerla, Liana Patel, Shu Liu, Tianneng Shi, Xiaoyuan Liu, Jared Quincy Davis, Emmanuele Lacavalla, Alessandro Basile, Shuyi Yang, Paul Castro, Daniel Kang, Joseph E. Gonzalez, Koushik Sen, Dawn Song, Ion Stoica, Matei Zaharia, and Marquita Ellis. 2025. Measuring Agents in Production. *arXiv preprint arXiv:2512.04123* (2025). doi:10.48550/arXiv.2512.04123

[20] Hannaneh B. Pasandi and Tamer Nadeem. 2026. The Reasoning Tax: Profiling Test-Time Compute Carbon in Agentic AI Workloads. In *Proceedings of the 5th Workshop on Sustainable Computer Systems (HotCarbon '26), ACM SIGENERGY Energy Informatics Review, Vol. 6, Issue 2*.

[21] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE.

[22] Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models. In *Proceedings of the 42nd International Conference on Machine Learning (ICML), Proceedings of Machine Learning Research, Vol. 267*. PMLR, 48371–48392.

[23] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large Language Model Connected with Massive APIs. In *Advances in Neural Information Processing Systems 38 (NeurIPS)*.

[24] You Peng, Youhe Jiang, et al. 2026. HexAGenT: Efficient Agentic LLM Serving via Workflow- and Heterogeneity-Aware Scheduling. *arXiv preprint arXiv:2605.16637* (2026).

[25] Yeonju Ro, Haoran Qiu, Íñigo Goiri, Rodrigo Fonseca, Ricardo Bianchini, Aditya Akella, Zhangyang Wang, Mattan Erez, and Esha Choukse. 2025. Sherlock: Reliable and Efficient Agentic Workflow Execution. *arXiv preprint arXiv:2511.00330* (2025).

[26] Mohamad Salim, Jasmine Latendresse, SayedHassan Khatoonabadi, and Emad Shihab. 2026. Tokenomics: Quantifying Where Tokens Are Used in Agentic Software Engineering. In *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR '26)*. arXiv:2601.14470.

[27] Ziji Shi, Chaoyi Ruan, Penghui Qi, Guangxing Huang, Xinyi Wan, Min Lin, and Jialin Li. 2025. Tetris: Efficient and Predictive KV Cache Offloading for Agentic and Reasoning Workloads. Preprint.

[28] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*.

[29] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, 173–191.

[30] The Apache Software Foundation. 2026. Apache Airflow. https://airflow.apache.org. Accessed August 2026.

[31] Noppanat Wadlom, Junyi Shen, and Yao Lu. 2026. Efficient LLM Serving for Agentic Workflows: A Data Systems Perspective. *arXiv preprint arXiv:2603.16104* (2026). doi:10.1145/3802046

[32] Cong Wang and Yusheng Zheng. 2026. Fork, Explore, Commit: OS Primitives for Agentic Exploration. *arXiv preprint arXiv:2602.08199* (2026).

[33] Xinglin Wang, Zishen Liu, Shaoxiong Feng, Peiwen Yuan, Yiwei Li, Jiayi Shi, Yueqi Zhang, Chuyi Tan, Ji Zhang, Boyuan Pan, Yao Hu, and Kan Li. 2026. On Time, Within Budget: Constraint-Driven Online Resource Allocation for Agentic Workflows. *arXiv preprint arXiv:2605.06110* (2026).

[34] Yuan Wang, Mingyu Li, and Haibo Chen. 2026. From Imperative to Declarative: Towards LLM-friendly OS Interfaces for Boosted Computer-Use Agents. In *Proceedings of the European Conference on Computer Systems (EuroSys '26)*. arXiv:2510.04607.

[35] Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. 2026. SWE-Pruner: Self-Adaptive Context Pruning for Coding Agents. *arXiv preprint arXiv:2601.16746* (2026).

[36] Joel L. Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. 2008. SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. In *Middleware 2008, Lecture Notes in Computer Science, Vol. 5346*. Springer, 223–242. doi:10.1007/978-3-540-89856-6_16

[37] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv preprint arXiv:2308.08155* (2023).

[38] Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. 2026. Reducing Cost of LLM Agents with Trajectory Reduction. *Proceedings of the ACM on Software Engineering* 3, FSE, Article FSE056 (2026). doi:10.1145/3797084

[39] Jiakai Xu, Tianle Zhou, Eugene Wu, and Kostis Kaffes. 2025. Toward Systems Foundations for Agentic Exploration. Preprint.

[40] Ya-Ting Yang and Quanyan Zhu. 2026. Toward Reliable Design of LLM-Enabled Agentic Workflows: Optimizing Latency-Reliability-Cost Tradeoffs. *arXiv preprint arXiv:2605.23929* (2026).

[41] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations (ICLR)*.

[42] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 521–538.

[43] Rui Zhang, Chaeeun Kim, and Liting Hu. 2026. A Policy-Driven Runtime Layer for Agentic LLM Serving. *arXiv preprint arXiv:2605.27744* (2026).

[44] Yusheng Zheng, Jiakun Fan, Quanzhi Fu, Yiwei Yang, Wei Zhang, and Andi Quinn. 2026. AgentCgroup: Understanding and Controlling OS Resources of AI Agents. *arXiv preprint arXiv:2602.09345* (2026).

[45] Yusheng Zheng, Yanpeng Hu, Tong Yu, and Andi Quinn. 2025. AgentSight: System-Level Observability for AI Agents Using eBPF. *arXiv preprint arXiv:2508.02736* (2025).

[46] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, 193–210.

[47] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *The Twelfth International Conference on Learning Representations (ICLR)*.