DAG-Aware Request Scheduling for LLM Agent Workloads Under External Rate Limits

Stanford CS244c

Berk Yalcinkaya[1] Filip Henriksson[2] Anushka De[3]

Department of Computer Science[1,2,3] Stanford University
{berkyalc, anushkad, filiph}@stanford.edu

Abstract

Modern LLM-powered agent systems accomplish complex tasks through multi-step workflows composed of interdependent LLM calls, often across many concurrent tenants competing for a shared organizational rate limit. Recent work [1] exploits DAG structure to schedule tasks on application-controlled inference clusters. However, this assumption breaks down when relying on external LLM APIs (e.g., OpenAI or Anthropic), where rate limits are opaque and scheduling visibility is limited.

In this work, we design a scheduling proxy deployed at the organization level to coordinate requests from multiple concurrent sessions to a single external LLM provider while respecting RPM and TPM constraints. Our design focuses on minimizing session makespan rather than individual request latency by exploiting lightweight structural signals observable at dispatch time. Experiments with real API workloads show that progress-aware scheduling can reduce end-to-end completion time relative to FIFO and exponential backoff, particularly under bursty demand.

1 Introduction

Recent advances in LLMs have enabled agentic AI systems that autonomously pursue goals through multi-step reasoning and execution [2]. Unlike traditional single request-response applications, these systems operate as iterative control loops: an orchestrator issues LLM calls, evaluates outputs, and dispatches further calls based on intermediate results [3, 4]. We refer to these dynamically expanding workflows as *agentic workloads*[3].

As applications increasingly deploy parallel agentic workloads to serve their users, scheduling workloads to minimize end-to-end completion time becomes a first-class concern. A system responsible for scheduling agentic workloads must grapple with contention for LLM serving capacity, the dependency structure within each workload, and the relative progress of competing sessions toward completion.

Parrot [1] demonstrates the value of DAG-aware scheduling for LLM workloads by introducing semantic variables that expose dependencies between calls, allowing the system to reconstruct the full request DAG and optimize execution, achieving up to 11.7× speedup. However, Parrot assumes the full workflow is known upfront and operates as the inference engine itself, scheduling directly on controlled GPU clusters. These assumptions do not hold for applications using external LLM providers such as OpenAI or Anthropic.

Relying on external LLM providers introduces resource contention concerns because these LLM services enforce rate limits. OpenAI, one of the most popular LLM providers, expresses their usage limits in terms of requests-per-minute (RPM) and tokens-per-minute (TPM) rate limits [5]. When these rate limits are enforced at the organization level, not per session, multiple organizational users sharing an API key draw from the same TPM and RPM budget without visibility into each other's consumption. After the request is submitted, the provider returns a binary signal: success or a 429 error. The standard mitigation is independent exponential backoff per session [6].

A natural improvement is to centralize dispatch through a shared FIFO queue, where all sessions submit requests and a single dispatcher drains the queue subject to the provider's rate limits. While this approach eliminates retry storms and improves rate-limit utilization, FIFO scheduling is agnostic to session structure and can introduce head-of-line blocking for multi-request workloads [7].

Our assumption of the nondeterministic nature of agentic workloads means the full task graph is never known in advance. However, an exploitable task structure still exists: An agent that dispatches parallel LLM requests must wait for all requests to complete before the next stage of reasoning can begin. Thus, a scheduler that tracks how many calls remain from each agentic workload can prioritize requests that are closest to unblocking downstream work for the agent, reducing session makespan without requiring knowledge of future phases. FIFO cannot make this distinction.

To address rate limiting and task structure ambiguity, we developed llmgateway.app, a scheduling proxy that

coordinates concurrent API calls across multiple agent workloads under a single OpenAI-imposed external rate limit. The proxy maintains a global view of in-flight requests across all workloads, continuously tracking RPM and TPM headroom while using each session's progress through its current phase to inform dispatch priority. We define our service-level objectives over session makespan, measured as mean and p95 end-to-end completion time, and evaluate three scheduler variants against FIFO and exponential backoff baselines on a concrete 11-call research pipeline under realistic rate limits against a live LLM provider.

2 Service Design

Our service is a scheduling proxy that sits between an organization's agent sessions and OpenAI, our chosen LLM provider. Clients create a session, submit LLM requests against that session, and receive completions while the proxy manages all dispatch ordering and rate limit enforcement. We assume that the organization implementing this service chooses an OpenAI model with a fixed token per minute and request per minute rate limit [5]. Users within this organization route all their LLM requests through the following minimal REST endpoints over HTTP/2:

Table 1: Minimal Proxy REST API Endpoints

Endpoint	Description
`POST /sessions`	Create a session. Returns a `session_id` that ties subsequent requests to one agentic workload.
`POST /call_types`	Register an LLM role. The proxy caches the system prompt and prepends it to later calls of this type.
`POST /sessions/{id}/completions`	Submit one LLM call to be dispatched to the provider.

Parameters.

- `POST /sessions`: none.
- `POST /call_types`: `name`, `system_prompt`.
- `POST /sessions/{id}/completions`: `call_type`, `messages`.

Internally, the proxy maintains a central request queue drained by a pluggable scheduler. On each drain step the scheduler selects the next request according to its policy consults a dual token-bucket rate limiter that enforces both RPM and TPM ceilings, and dispatches the request to the OpenAI API only when capacity is available via the `try_acquire()` method. Our rate limiter enforces RPM and TPM simultaneously via two independent token buckets, each refilling continuously at limit / 60 per second. OpenAI requires that in-flight requests reserve capacity specified by the user. Our schedulers differ in their estimation, but for all scheduling schemes, actual token usage reported by the provider is fed back to the rate limiter to reconcile over- or under-estimates via the `reconcile()` endpoint.

We implemented this service using Python FastAPI with 2,216 lines of code on a GCP `e2-medium` instance (2 vCPUs, 4 GB memory).

3 Defining a test workload

To evaluate our scheduler, we define a research agent as a representative agentic workload. The agent operates as an orchestrator loop with access to three classes of LLM-backed workers: analysts (which retrieve and analyze information on subtopics), a synthesizer (which aggregates findings into a coherent draft), and reviewers (which evaluate the draft for factual accuracy, citation quality, and style). In its default form, the orchestrator dynamically decides at each iteration which workers to invoke, how many to dispatch in parallel, and whether to loop back for additional rounds of analysis or review, producing a nondeterministic task graph that varies across sessions. The system prompts that define each type of LLM call were designed to induce varied output lengths. For instance, `analyst_statistical` produces short bulleted lists (≈ 100 tokens), while `analyst_deep_analysis` produces exhaustive analyses ($\approx 1,500$ tokens), creating natural variance in completion times within the analyst fan-out group.

For evaluation, nondeterminism can confound scheduler comparisons. We therefore fix the pipeline to a deterministic strict mode with two fan-out/fan-in stages: the orchestrator issues a planning call, fans out to five analysts in parallel, synthesizes their outputs, fans out to three reviewers in parallel, and produces a final synthesis, for a total of 11 LLM calls per session with two barrier synchronization points. This ensures all sessions generate identical load, isolating the effect of dispatch ordering on session makespan. Despite fixing the workload structure, the scheduler receives no structural metadata beyond the number of currently in-flight calls per session (see Section 2).

This workload structure exposes barrier synchronization bottlenecks, where the completion time of a fan-out stage is determined by the slowest call rather than the average. A single straggling request can delay the entire session and all downstream work. Addressing these straggler effects is the primary goal of our scheduler.

4 Simulation Design

Using our API, we built a simulation framework that runs 30 concurrent sessions through our gateway service. Prompts and arrival times are pre-generated with a fixed random seed, so every scheduler faces the exact same

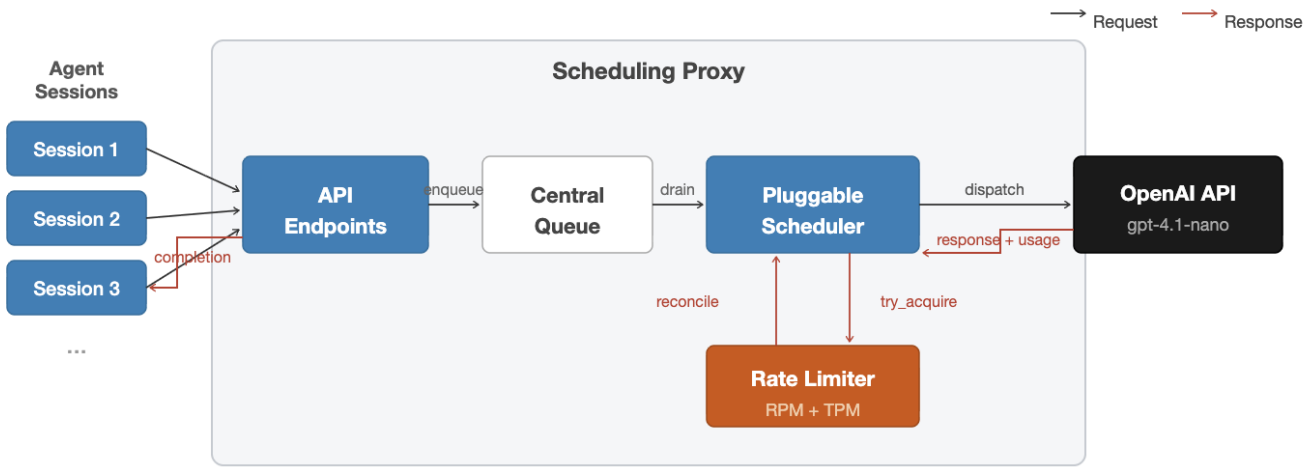

Figure 1: Concurrent sessions submit to a central scheduler, which dispatches to the rate-limited API.

requests in our simulations. All experiments use real API calls to `OpenAI GPT-4.1-nano` rather than synthetic latencies, with each session running the workload discussed in Section 3 in strict mode (exactly 11 LLM calls).

For the purpose of refining scheduler designs, we ran experiments with artificial rate limits to control simulation cost, enabled by our `RateLimiter`'s support for arbitrary TPM and RPM values.

Given 30 sessions, each dispatching 11 requests, we derive two rate-limit profiles that create distinct bottleneck regimes:

- **RPM-heavy** (`RPM`=20, `TPM`=200K): request capacity is the primary bottleneck.
- **TPM-heavy** (`RPM`=60, `TPM`=40K): token capacity is the primary bottleneck.

For each rate-limit profile, we additionally evaluate two session start-time regimes to test whether schedulers behave differently under steady versus bursty load:

- **Constant:** sessions begin sending requests at uniform intervals (every 4 s) to model steady usage near the rate limit.
- **Bursty:** clusters of 2–5 sessions start 0.5–2 s apart, separated by 15–35 s gaps, modeling interactive and correlated demand spikes.

For each scheduler design (Section 5), we run 30 concurrent sessions across all four combinations of bottleneck regime and timing regime. Between scheduler runs, the rate limiter is reset with a 30 s cooldown. The full experimental matrix (4 scenarios $\times$ 5 schedulers) yields 20 runs, which we aggregate into a combined comparison.

5 Scheduler Design

Baselines. We evaluate against two standard approaches. *Exponential backoff* uses no centralized queue: each session retries independently with doubling delays and jitter. This is simple but creates thundering-herd effects under shared rate limits, producing 6$\times$ more RPM throttles than queue-based alternatives in our experiments. *FIFO queuing* centralizes dispatch and eliminates retry storms, but enforces strict arrival order with no session awareness, causing head-of-line blocking when a large request at the front of the queue prevents smaller, dispatchable requests from proceeding.

Call-level metrics failed. We explored several per-request priority signals before converging on the structural MapReduce approach. Shortest-job-first by input token count proved unreliable: prompt length is a poor predictor of response time. Learned output-token estimates via an exponential moving average offered a better signal but still suffered head-of-line blocking. Predicted wall-clock duration was too noisy due to network and queuing variance. These experiments confirmed that no call-level metric reliably serves as a primary scheduling signal, and that optimizing individual request ordering misses the true objective: minimizing session makespan.

MapReduce priority. Our core scheduling signal is structural rather than per-request:

$$\text{priority}(s) = \frac{1}{n_s},$$

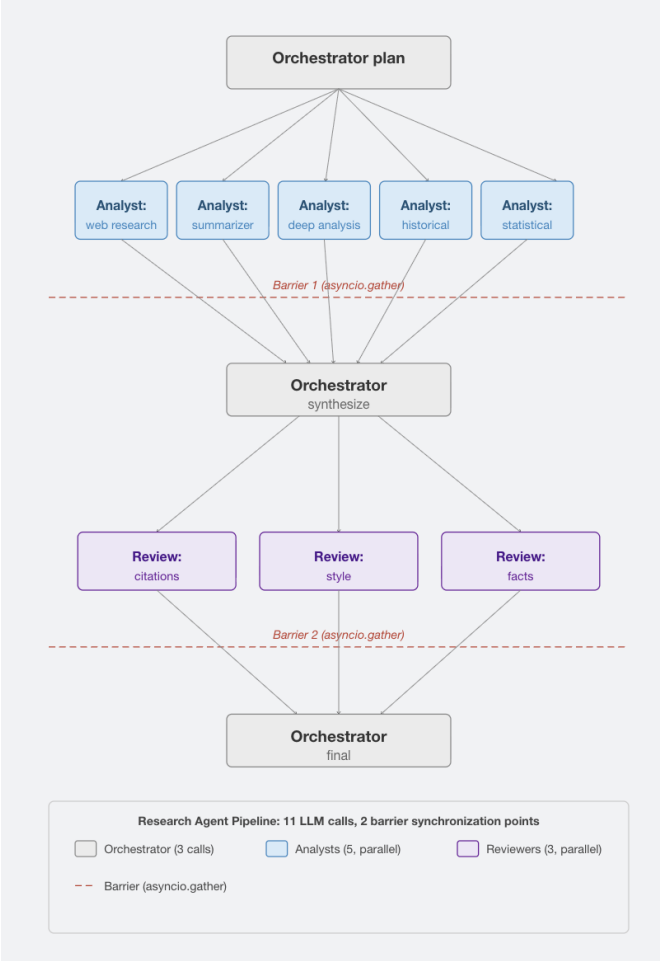

Figure 2: Orchestrator plans the research (1 LLM call), 5 analysts in parallel (5 LLM calls); barrier until all 5 complete, Orchestrator synthesizes findings into a draft (1 LLM call), 3 reviewers in parallel (3 LLM calls); barrier. Orchestrator final synthesis (1 LLM call).

where n_s denotes the number of in-flight requests for session s.

The session identifier tracks in-flight calls: submission increments the count, and completion decrements it. Because agentic pipelines are barrier-synchronized, this count directly reflects the current fan-out width. As parallel calls within a phase complete, the remaining calls receive progressively higher priority: five analysts begin at priority 0.2, and the final straggler blocking the barrier reaches priority 1.0. This ensures the call whose completion most advances end-to-end progress is dispatched first. Priorities are recomputed on every dispatch decision; ties break by arrival order.

Per-call-type token estimation. Although call-level metrics failed as the primary signal, per-call-type output token learning proved valuable as a secondary mechanism within our MapReduce-priority scheme. Each LLM role produces roughly consistent output sizes; for example, a statistical analyst may generate roughly 100 tokens, while a deep analyst may generate roughly 1,500. A per-call-type exponential moving average ($\alpha = 0.3$) learns these distributions within a few observations, enabling tighter in-flight TPM reservations to OpenAI.

Adaptive tie-breaking. Within a fan-out group, all calls share the same MapReduce priority. The adaptive variant breaks ties by dispatching shorter predicted outputs first, freeing rate-limiter capacity sooner and allowing subsequent calls to proceed without waiting. The composite priority key is

$$\big(-\text{priority}(s),\ \text{predicted\_tokens},\ \text{arrival\_order}\big),$$

so the structural signal remains dominant, with output prediction operating only within a priority tier.

TPM-aware skipping. When the highest-priority call exceeds the available TPM budget, strict priority ordering would idle the entire queue. The skip variant instead dispatches the highest-priority call that fits the current budget, avoiding wasted capacity. A bisect-sorted index enables $O(\log n)$ budget queries. Combined with per-call-type EMA estimates, this reduces idle time under TPM-constrained workloads without violating the MapReduce priority ordering for calls that do fit.

6 Simulation Results

We evaluate five schedulers across four scenarios (2 rate-limit profiles × 2 arrival patterns), each running 30 concurrent sessions of 11 LLM calls against a live OpenAI endpoint. Table 2 reports mean and P95 session times.

RPM-heavy regime. Under RPM constraints, MapReduce achieves ≈34–35% mean reduction over FIFO in both arrival patterns (34.3% constant, 35.1% bursty). All three MapReduce variants perform nearly identically (≈480 s constant, ≈425 s bursty), confirming that skip and adaptive tie-breaking add negligible value when TPM is abundant. Backoff produces 6× more throttle attempts than queue-based alternatives (1,798 vs. ≈300), and under bursty arrivals it outperforms FIFO (600.92 s vs. 647.95 s) because simultaneous request clusters cause severe head-of-line blocking in the FIFO queue.

TPM-heavy regime. Base MapReduce maintains 18–22% improvement over FIFO (21.6% constant, 17.6% bursty). However, the skip variants regress dramatically:

Table 2: Session completion times across all scenarios. Bold marks the best mean and P95 per scenario.

	RPM-heavy, Constant		RPM-heavy, Bursty		TPM-heavy, Constant		TPM-heavy, Bursty	
Scheduler	Mean	P95	Mean	P95	Mean	P95	Mean	P95
Backoff	744.78	876.59	600.92	888.08	897.19	1,096.67	870.14	1,125.04
FIFO	728.88	831.05	647.95	800.39	825.53	1,114.29	720.07	**953.98**
MapReduce	**478.79**	828.01	**420.84**	709.75	**647.47**	**1,085.11**	**593.66**	999.83
MR Skip	480.59	**811.89**	426.11	**702.08**	932.11	1,160.48	864.07	1,091.60
MR Skip Adaptive	482.52	823.38	430.22	702.47	918.89	1,152.33	885.24	1,137.20

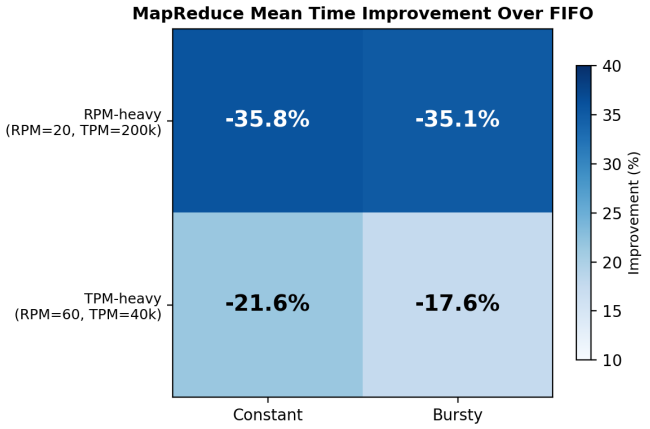

Figure 3: MapReduce mean session time improvement over FIFO across all scenarios.

MR Skip is 44% worse than base MapReduce under constant arrivals and worse than both baselines. The cause is systematic token overestimation — the per-call-type EMA with a 20% safety buffer produces a ≈1.31× overestimation ratio, causing the budget check to falsely reject high-priority calls. Orchestrator calls, which have the highest estimated token counts due to accumulated context, are most likely to be skipped, yet these are the critical-path calls whose completion unblocks downstream work. Consequently, our simulations are unable to validate the skip extension under TPM-constrained conditions, and the adaptive tie-breaking extension proved negligible relative to the base MapReduce signal.

Table 3: MapReduce Improvement Over Baselines

Scenario	**MR Mean (s)**	**vs. FIFO**	**vs. Backoff**	**Skip?**
RPM, constant	478.79	−34.3%	−35.7%	Negligible
RPM, bursty	420.84	−35.1%	−30.0%	Negligible
TPM, constant	647.47	−21.6%	−27.8%	Harmful
TPM, bursty	593.66	−17.6%	−31.8%	Harmful

Cross-scenario summary. As Table 3 and Figure 3 show, gains are larger under RPM-heavy conditions (≈34–35%) than TPM-heavy (≈18–22%): when every request slot is scarce, priority ordering has maximum impact. Bursty arrivals amplify P95 benefits because inter-burst gaps let near-completion sessions drain without contention. TPM-aware skipping is harmless when token budget is abundant but produces 40%+ regression under TPM constraints, indicating that skip-based scheduling requires substantially more accurate token estimation to be viable.

The artificial rate limits above allowed rapid iteration over scheduler designs at low cost. We now validate whether the observed patterns hold under production-tier rate limits with real API calls at scale

7 Results under Real Rate Limits

To validate our simulation findings, we evaluated FIFO and MapReduce under OpenAI's production-tier rate limits (5,000 RPM, 2,000,000 TPM) using `gpt-4.1-nano`, launching 200 concurrent sessions through our scheduling proxy with a fixed seed ensuring identical workloads. We tested three arrival regimes: constant at 0.1 s and 0.25 s intervals, and bursty at 0.25 s spacing with wave-like clusters of 10–30 sessions separated by quiet gaps of 3–6 times the burst duration.

Figure 4 reports mean and P95 session latency. At 0.25 s constant spacing, sessions are spread out enough that queue depth remains shallow and MapReduce's priority signal provides no benefit; FIFO achieves lower mean latency (94.1 s vs. 99.1 s). To confirm this is due to insufficient contention, we tightened the arrival interval to 0.1 s, increasing queue pressure. Under this denser regime, MapReduce recovers a 9% mean improvement (106.0 s vs. 116.5 s), confirming that the structural signal requires meaningful queue depth to outperform FIFO. The effect is strongest under bursty arrivals, where MapReduce achieves a 15% mean reduction (82.6 s vs. 97.6 s) and a modest P95 improvement (139.9 s vs. 143.4 s). Across all three regimes, MapReduce's advantage scales with contention: negligible when queues are shallow, and substantial when burst-induced queueing creates dispatch bottlenecks. TThis is consistent with the simulation results in Section 6

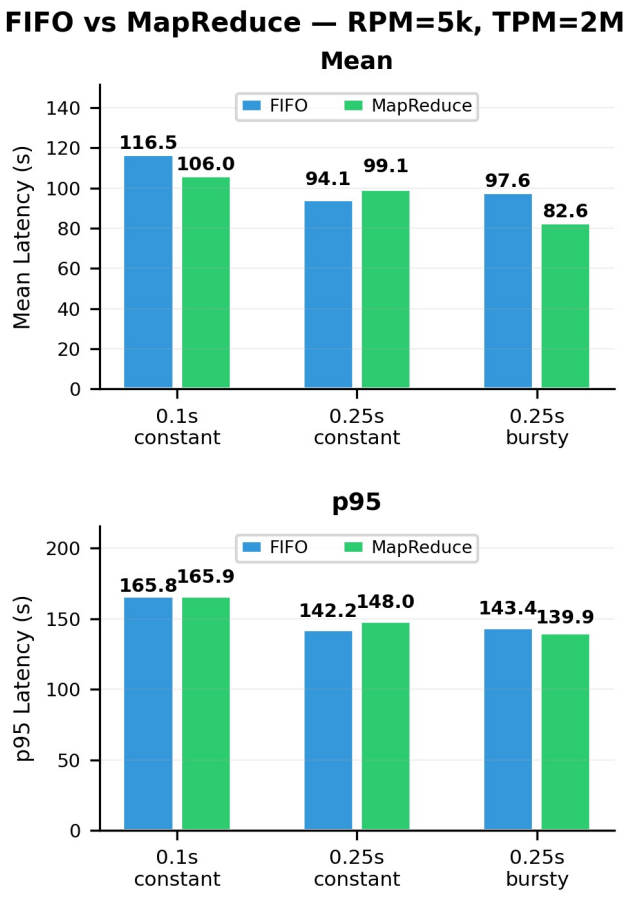

Figure 4: FIFO vs. MapReduce under production-tier rate limits (200 sessions, RPM=5,000, TPM=2M). Mean and P95 session latency across three arrival regimes.

8 Conclusion

Agentic LLM workloads introduce a new scheduling challenge: multi-stage, barrier-synchronized request DAGs competing under opaque external rate limits. While centralized FIFO dispatch improves utilization over independent exponential backoff, it fails to account for session structure and can prolong end-to-end completion time due to head-of-line blocking and straggler effects.

In this work, we presented a lightweight scheduling proxy that exploits a single observable signal, the number of in-flight calls per session, to approximate progress toward barrier completion. Under controlled rate limits, our MapReduce-inspired priority scheme achieves 18–35% mean session time reductions over FIFO and backoff baselines across both RPM-constrained and TPM-constrained regimes. Under production-tier rate limits (5,000 RPM, 2M TPM) with 200 concurrent sessions, MapReduce provides up to 15% mean improvement under bursty arrivals, with gains scaling with queue contention; when queue depth is shallow, the structural signal offers no advantage over FIFO.

More broadly, our results suggest that effective scheduling for agentic workloads does not require full workflow visibility, but rather simple progress signals that approximate downstream impact. As agentic systems continue to scale, we believe structure-aware scheduling layers will become an essential part of production-level LLM infrastructure. Future work includes extending these ideas to heterogeneous model routing, multi-provider dispatch, and learning-based scheduling policies.

References

[1] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable, 2024. URL https://arxiv.org/abs/2405.19888.

[2] Nadia Sibai, Yara Ahmed, Serry Sibaee, Sawsan Al-Halawani, Adel Ammar, and Wadii Boulila. The path ahead for agentic ai: Challenges and opportunities, 2026. URL https://arxiv.org/abs/2601.02749.

[3] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

[4] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

[5] OpenAI. Rate limits. https://developers.openai.com/api/docs/guides/rate-limits/, 2026. Accessed: 2026-03-14.

[6] Behrooz Farkiani, Fan Liu, and Patrick Crowley. Rethinking http api rate limiting: A client-side approach, 10 2025.

[7] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Queue management for slo-oriented large language model serving, 2025. URL https://arxiv.org/abs/2407.00047.