Now or Later: Scheduling Agent Inference Across Real-Time and Batch to Meet SLOs at Lower Cost

Tarun Gopinath
Amazon
tarungop@amazon.com

Atul Kulkarni
Amazon
atukulka@amazon.com

Shrikar Katti
Amazon
shrikar@amazon.com

Abstract

An agent inference can execute immediately through a real-time streaming API or be deferred to a batch API that cloud providers price at approximately half the real-time rate. Existing agent interfaces generally bind a request to one mode before execution. The runtime therefore cannot revise the mode as conditions change. We identify three requirements for per-inference mode scheduling: (i) the agent definition must abstract the inference mode; (ii) agent execution must be decoupled from the client-facing API; and (iii) a scheduler must adapt the mode dynamically to latency and cost objectives. We present CONDUCTOR, a design that satisfies these requirements, and evaluate its scheduler in simulation using parameterized per-call latency and cost profiles. Given an SLO, the scheduler selects an operating point on the cost–attainment frontier. With exact step-count estimates, an SLO-optimized policy meets every simulated deadline at approximately 21% lower cost than all-immediate execution (cost 0.79; deadline-feasible oracle cost 0.64). When a submitted batch call runs long, the policy launches a duplicate real-time stream at the latest safe time; both requests are charged. The resulting rescue overhead is tunable, decreasing from 13.7% at the cost-minimizing p50 setting to 1.1% at p90. A no-rescue variant trades SLO attainment for lower cost. The task interface supplies an end-to-end SLO, and the runtime selects the execution mode.

1 Introduction

An LLM agent is a workflow that interleaves model inferences, tool calls, and control flow [16]. Production runtimes commonly provide several ways to execute an inference. We focus on two modes at opposite ends of the cost–latency trade-off. **Immediate** inference uses a synchronous, token-streaming API to minimize time to first token, but incurs the full real-time price and consumes bounded online capacity. **Delayed** inference uses a provider batch API that accumulates requests before submission. It may take minutes or hours and require a minimum batch size, but cloud providers offer it at approximately a 50% discount relative to real-time inference [2, 3, 13].

This discount is the primary cost lever studied here. In self-hosted systems, the same scheduling flexibility can instead improve utilization by moving deferrable work into batches and periods of low interactive load [14] (§7).

In the provider APIs we consider, clients select between these modes through separate interfaces [2, 13]. The selected mode remains fixed for the request. This interface asks the client to choose an execution mechanism rather than state the task's latency requirement. We propose that each agent task carry an explicit end-to-end service-level objective (SLO). The runtime can then combine this target with its own load, batch-path state, and observed provider turnaround history to select the execution mode for each inference call.

40th Conference on Neural Information Processing Systems (NeurIPS 2026). Workshop: AgenticOS.

We argue that **execution mode should be a per-inference scheduling decision** made by a scheduler with system-wide visibility. This model requires three conditions:

- **(i) The agent definition abstracts execution mode without changing workflow semantics (§2).** The scheduler can select immediate or delayed execution at runtime while preserving the workflow's prompts, tool behavior, and control flow.
- **(ii) Execution is decoupled from the client API (§3),** allowing the runtime to suspend and resume work without holding a client connection open.
- **(iii) The scheduler adapts the mode dynamically (§4),** dividing one end-to-end SLO across dependent calls and revising decisions as execution proceeds.

We present CONDUCTOR, a design that satisfies all three requirements, and evaluate its scheduler in simulation using parameterized per-call profiles (§5). The scheduler requires an explicit SLO supplied by the client or deployment.

2 Mode-Portable Agentic Execution Model

We implement agents using the typed Java dataflow DSL described in prior work [7]. Agent authors chain type-safe combinators. These combinators construct an immutable, mode-neutral workflow graph, and typed edges carry agent state between operations. The Java type system checks each operation's input and output contract as the graph is assembled. Consequently, execution mode is absent from the agent definition. Mode portability requires both execution paths to preserve the same prompts, tool behavior, control flow, and typed values. Otherwise, selecting a mode would change the workflow rather than only its execution strategy.

The graph is built from a small set of combinators. `map` applies stateless transformations such as prompt construction, response parsing, and tool dispatch; `iterate` expresses bounded loops such as ReAct [16]; `branch` and `parallelBranch` express conditional and parallel fan-out; and `compose` delegates to another registered agent. The key operator is `infer`. It is a first-class graph node rather than an opaque model call hidden inside `map`. Each `infer` node declares a prompt, model configuration, and typed result. This node creates a suspension boundary where the runtime can select an execution strategy without changing the graph's functional semantics.

CONDUCTOR uses this boundary to choose an implementation independently for every inference call. An immediate decision invokes the streaming interface inline and forwards tokens as they arrive. A delayed decision checkpoints the graph state, submits the inference through the batch path, correlates the eventual result with the suspended invocation, and resumes the same continuation. Thus, the DSL separates *what* the agent computes from *when and how* each model call executes. Our implementation uses this typed dataflow runtime. The same scheduling design could apply to frameworks that expose inference boundaries and keep execution mode out of the agent definition.

3 Decoupling Execution from the Client

CONDUCTOR is integrated with the runtime (Figure 1). Its decision API accepts the request SLO and system state and returns IMMEDIATE or DELAYED; the runtime executes the selected mode. Delayed execution requires decoupling both the connection and workflow progress from the client-facing API.

Connection decoupling. Clients submit requests under stable session identifiers over WebSocket [6]. Session and result state are independent of the socket, and sequence-numbered output allows reconnection from the last acknowledged cursor. Deferred work can therefore complete without holding a connection open or exposing its execution mode.

Per-step decisions. At each inference boundary, the runtime calls `decide(request, stepContext)`. Immediate calls execute inline and stream tokens. Delayed calls atomically checkpoint workflow state, enter the batch inference path described in prior work [7], and resume when their results arrive without re-executing prior steps. We treat that path as opaque except for its commitment state and result callback.

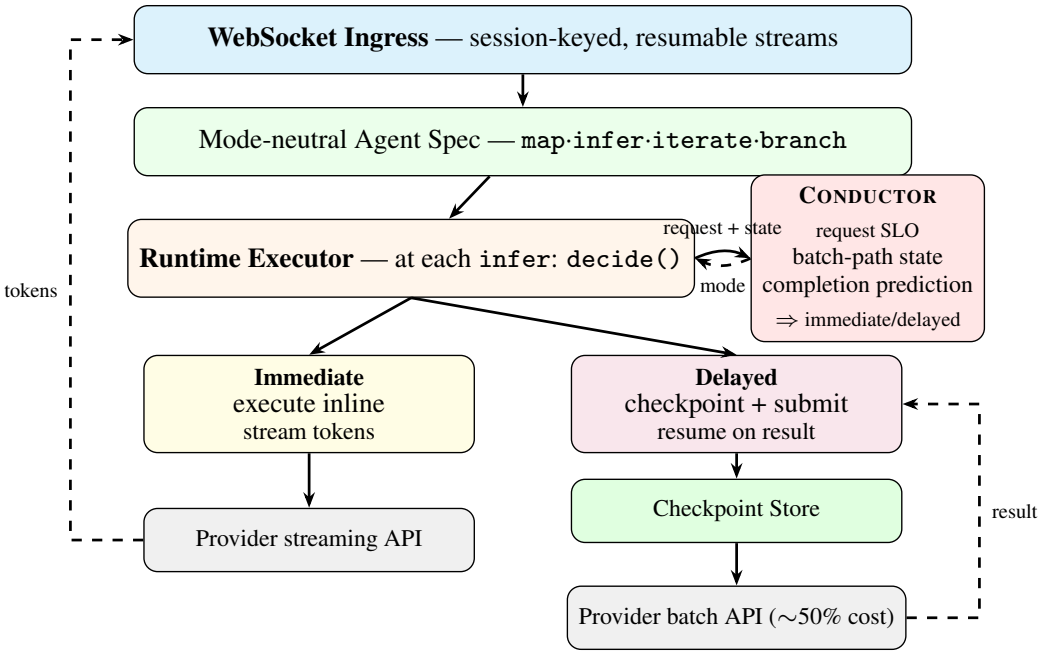


Figure 1: Architecture. At each inference step, CONDUCTOR selects immediate streaming or checkpoint-and-batch execution from the request SLO and current system state. The runtime executes the selected mode.

Post-commitment rescue. A committed batch request cannot be converted back to immediate execution. Depending on the configured rescue policy, the runtime can start a duplicate real-time request if the result has not arrived by the latest safe time. The first usable result resumes the workflow, but the provider charges for both requests. This fallback limits misses caused by batch-tail latency, but increases cost.

4 Adapting the Mode Dynamically

The scheduler operates on an *agent job*: one execution that issues a sequence of dependent inference calls under a single end-to-end SLO. Call c_{i+1} cannot start until c_i returns, so batching an intermediate call delays every subsequent call. The scheduler must allocate one deadline budget across a sequence whose final length is not known in advance. It first divides the budget across the estimated sequence (§4.1) and then assigns each call a mode under that budget and the selected cost objective (§4.2).

4.1 Budgeting Across the Inference Sequence

At job start, the scheduler estimates the number of inference calls, $\hat{k}$, from the *agent profile*. The profile captures the agent type, configuration, and expected orchestration shape. A single-shot RAG agent [9] may issue one inference, whereas a ReAct loop [16] may issue a variable number based on intermediate tool results and its stopping condition. A deployment can derive conservative estimates from static bounds (e.g., maximum iterations) or execution history; unseen tasks can default to an upper bound or immediate execution. Given deadline T, start time t_0, and estimate $\hat{k}$, the scheduler assigns each call a *per-step sub-deadline*. At step i, issued at time t with $r = \hat{k} - i$ estimated calls remaining (with i starting at zero), the sub-deadline is

$$d_i \;=\; t \;+\; \frac{(t_0 + T) - t}{\max(1, r)}.$$

This equation divides the remaining budget equally among the estimated remaining steps. The decision is therefore both call-specific and job-aware: a step is batchable only if its budget can

absorb batch latency while preserving sufficient time for subsequent calls. Equal division is a simple heuristic, not an optimal allocation; the mechanism also supports nonuniform budgets derived from predicted inference and tool latencies.

The scheduler recomputes these terms after each call. A job that consumes budget faster than expected or exceeds its estimated sequence length shifts toward immediate execution for its remaining steps.

4.2 Scheduling Policy

Admission. A call executes IMMEDIATE when its sub-deadline cannot accommodate the predicted batch turnaround. Otherwise, it enters the delayed path. Because the scheduler recomputes the budget after every step, jobs running behind schedule shift toward immediate execution; jobs ahead of schedule can defer more calls.

Pre-submission diversion. Before the delayed path reports a call as committed, the scheduler rechecks its deadline against a selected turnaround percentile plus a safety margin. Calls that no longer have enough time are diverted to the real-time API. After commitment, the selected mode is fixed.

Post-commitment rescue. After commitment, the scheduler may invoke the rescue mechanism described in §3 at the latest time that can still meet the sub-deadline. Enabling rescue trades duplicate cost for protection against batch-tail latency; disabling it permits deadline misses.

Decision quality depends on two predictions. The per-job step count $\hat{k}$ determines how the job budget is divided (§4.1). Underestimating it gives early calls too much time and can starve later calls; overestimating it reduces potential savings.

The second prediction is provider-controlled batch turnaround, which has high variance. CONDUCTOR maintains an empirical distribution for each model and batch size, schedules against a selected percentile plus margin, and updates the distribution as calls complete. When turnaround variance consumes the available budget, only jobs with looser deadlines benefit from batching. §5 evaluates sensitivity to both predictions.

5 Evaluation

We evaluate three questions: whether per-call mode selection reduces cost while preserving an end-to-end deadline; how scheduling percentile and rescue change the cost–attainment trade-off; and how step-count error affects the result. We compare CONDUCTOR with policies that bind execution mode before a job begins, reflecting the interface limitation that motivates this work. These baselines isolate the benefit of revisable decisions rather than represent every adaptive scheduler; the oracle quantifies headroom for stronger allocation policies.

5.1 Experimental Setup

Workload. A seeded discrete-event simulator executes streams of agent jobs. Each job contains dependent inference calls governed by one end-to-end deadline. We normalize tightness as $R = \text{deadline}/(k \times \text{batch p50 per call})$, where k is the true call count. Small R permits few delayed calls; larger R permits more. We sample R log-uniformly from 0.1 to 30; approximately one third of jobs fall in the partially batchable range $R \approx 0.5$–3. Jobs arrive as a Poisson process at 0.5 jobs/s and use a Sonnet-class model with 10K-token context. A job has $k \in \{1, 2, 3, 5, 8\}$ calls (mean approximately 3), each producing 200–2,000 tokens. The delayed-path model uses batch sizes of 8–64 records and up to 120 s of pre-commitment delay. The headline comparison and percentile sweep use exact step counts ($\hat{k} = k$), 32 replicas of 2,000 jobs, and 95% confidence intervals.

Latency and cost model. An immediate call takes $\mathrm{TTFT}(\text{context}, \text{tier}) + n_{\text{out}}/\text{speed}(\text{tier})$. Output speed and time to first token come from published per-model measurements [4]: approximately 40–90 tokens/s and 0.8–2.9 s TTFT at 10K context. A delayed call resamples turnaround measurements collected from evaluation runs in prior work [7]. The empirical distribution is heavy-tailed, with

a median of approximately 49 minutes and p90 of approximately 7 hours. We normalize cost to one immediate call: immediate costs 1.0, delayed costs 0.5, and a rescued call costs 1.5 because both requests are charged. The controls are the batch-turnaround percentile and whether to rescue a late batch result. Here p50 and p90 denote the empirical batch-turnaround quantile used as the completion-time prediction.

Tool calls. The simulator does not model tool calls as separate events. Instead, the per-step model treats nominal tool overhead as part of its latency variation. The evaluation thus applies to inference-dominated workflows, not workloads with substantial, heavy-tailed, rate-limited, or failure-prone tools.

Scope and assumptions. This simulation evaluates the scheduling policy rather than a production guarantee. It assumes admitted real-time requests follow the modeled latency and omits provider throttling, contention, failures, and rate limits. These effects consume deadline slack and require deployment-specific calibration; rescue cannot repair immediate-path unavailability.

Policies. ALL-IMMEDIATE and ALL-DELAYED establish the latency and cost extremes. STATIC fixes half the jobs to each mode using a coarse class label independent of the deadline. BATCH-BATCHABLE assigns each job to one mode according to whether $\hat{k}$ calls fit under the selected batch percentile. DEADLINE-AWARE decides per call and rescues late delayed calls; DEADLINE-AWARE-COST disables rescue. An oracle with perfect realized latencies provides the minimum-cost deadline-feasible assignment.

Deadline attainment and cost. Table 1 shows the main result. In the headline sweep, the cost-minimizing DEADLINE-AWARE setting meets every simulated deadline at relative cost 0.791: approximately 21% below ALL-IMMEDIATE, although still above the 0.642 deadline-feasible oracle cost. It is the only non-oracle policy among those evaluated that combines 100% SLO attainment with a cost below 1.0.

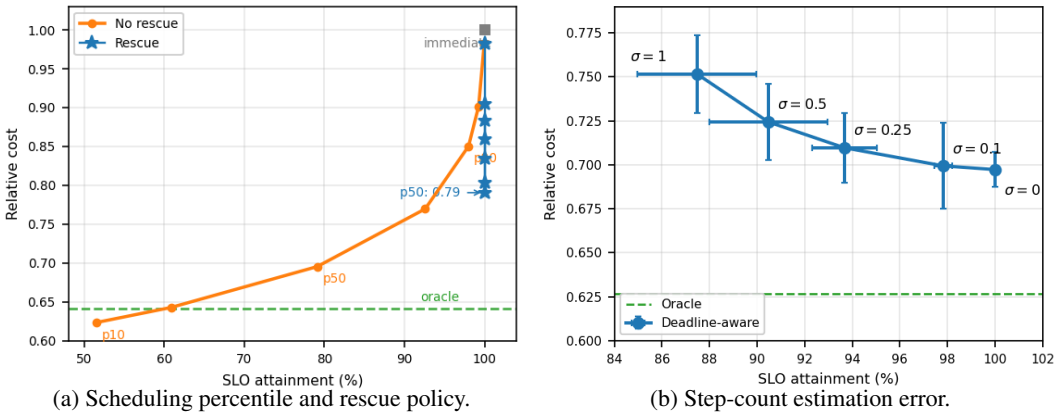
Assigning one mode to an entire job leaves savings on the table or misses deadlines. BATCH-BATCHABLE lowers cost to 0.702 but attains only 85.1% of deadlines. At the same p50 setting, no-rescue DEADLINE-AWARE-COST costs slightly less (0.695) but attains only 79.1%. DEADLINE-AWARE closes this attainment gap by starting a duplicate real-time call at the latest safe time when a submitted batch result is overdue.

Table 1: Cost and SLO attainment at p50 and p90 (95% CI; $32\times2{,}000$ jobs). Cost is relative to all-immediate. Figure 2(a) shows the full percentile sweep.

Policy (setting)	SLO attainment (%)	Rel. cost
ALL-IMMEDIATE	100.0	1.000
ALL-DELAYED	49.3 ± 0.3	0.500
STATIC	74.5 ± 0.4	0.750 ± 0.002
BATCH-BATCHABLE	85.1 ± 0.2	0.702 ± 0.002
ORACLE (feasible bound)	100.0	0.642 ± 0.001
DEADLINE-AWARE — *SLO-optimized (rescue)*		
at p50 (cost-min)	100.0	0.791 ± 0.002
at p90	100.0	0.860 ± 0.002
DEADLINE-AWARE-COST — *no rescue*		
at p50	79.1 ± 0.3	0.695 ± 0.002
at p90	98.0 ± 0.1	0.850 ± 0.002

Selecting an operating point. Figure 2(a) sweeps the batch-turnaround percentile. Without rescue, a more conservative percentile raises attainment from 79% to 98%, but raises relative cost from 0.70 to 0.85 by executing more calls immediately.

With rescue enabled, attainment remains 100% throughout the sweep. Cost is U-shaped and reaches its minimum of 0.79 at p50, with 13.7% rescue overhead. Overhead falls to 1.1% at p90 and 0.4% at p95 as conservative admission sends more calls immediately. These controls span lower-cost best-effort to 100% attainment in this workload.



(a) Scheduling percentile and rescue policy.

(b) Step-count estimation error.

Figure 2: Cost–attainment trade-offs. (a) Rescue protects attainment; no rescue favors cost. (b) Step-count error lowers attainment and raises cost. Error bars show 95% confidence intervals.

Sensitivity to step-count error. The headline result assumes $\hat{k} = k$. A separate sweep uses a fixed 14,645 s deadline (approximately five batch medians) and perturbs $\hat{k}$. With a known step count, DEADLINE-AWARE attains 100% at relative cost 0.697. Increasing unbiased lognormal error from $\sigma = 0$ to 0.25 lowers attainment to 93.7% and raises cost to 0.710; at $\sigma = 1.0$, attainment falls to 87.5% and cost rises to 0.752. Thus, at $\sigma = 0.25$, the policy remains 29% cheaper than all-immediate but no longer preserves the headline 100% attainment.

Direction also matters. At fixed $\sigma = 0.25$, a log-space bias of -0.7 reduces the mean estimate from 3.06 to 1.64 calls and lowers attainment to 71.1%. A bias of $+0.7$ raises the mean estimate to 6.38 calls and restores 100% attainment at relative cost 0.787, retaining approximately 21% savings over all-immediate. Underestimation gives early calls too much budget and can starve calls omitted from the estimate. Rescue cannot repair this end-to-end budget error. Conservative overestimation preserves more deadline margin by sending more calls to the immediate path. The headline cost-and-attainment claim therefore depends on exact or sufficiently conservative step-count estimates: overestimation sacrifices savings, whereas underestimation can exhaust slack before unanticipated calls execute. Figure 2(b) shows this trade-off.

6 Related Work

Agent and SLO-aware scheduling. Autellix schedules and preempts calls within agent programs [11], Parrot exposes cross-request dataflow [10], and Teola optimizes workflow graphs [15]. DistServe [19] and JITServe [18] target request-level SLOs using online-serving mechanisms such as continuous batching [17], PagedAttention [8], and chunked prefill [1]. These systems optimize call order or online execution; CONDUCTOR instead selects between immediate and delayed modes for each call under a job deadline.

Cost-aware and hybrid inference. Routing and cascading reduce cost through model selection [5, 12], whereas CONDUCTOR preserves the model and changes its execution mode. Hybrid offline–online schedulers mix independent batch and interactive requests at the cluster level [14]. These approaches are complementary to per-call mode selection within an agent job.

7 Conclusion

CONDUCTOR makes execution mode a per-call decision under an agent job's deadline. With exact step counts, it meets every simulated deadline at 21% below all-immediate cost. Moderate unbiased error ($\sigma = 0.25$) retains a 29% reduction but lowers attainment to 93.7%; conservative overestimation restores 100% at approximately 21% savings. The headline result therefore requires exact or conservative estimates. Provider-hosted systems can capture batch discounts; self-hosted systems could instead improve utilization [17, 14]. Future work includes production validation with tool and provider-capacity effects.

References

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. SARATHI: Efficient LLM inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[2] Amazon Web Services. Amazon Bedrock batch inference. `https://docs.aws.amazon.com/bedrock/latest/userguide/batch-inference.html`, 2024.

[3] Amazon Web Services. Amazon Bedrock pricing. `https://aws.amazon.com/bedrock/pricing/`, 2026. Accessed August 2026.

[4] Artificial Analysis. Artificial analysis: Llm performance leaderboard (output speed and time-to-first-token). `https://artificialanalysis.ai`, 2025. Per-model output tokens/s and TTFT measurements.

[5] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

[6] Ian Fette and Alexey Melnikov. The WebSocket protocol. RFC 6455, Internet Engineering Task Force, 2011.

[7] Tarun Gopinath, Atul Kulkarni, Vijay Rajakumar, Shrikar Katti, and Parthasarathy Govindarajen. Substrate-portable execution for production LLM workflows. *arXiv preprint arXiv:2609.06128*, 2026.

[8] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 23)*, 2023.

[9] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020.

[10] 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. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[11] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wu, Xiaocheng Lin, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Autellix: An efficient serving engine for LLM agents as general programs. *arXiv preprint arXiv:2502.13965*, 2025.

[12] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

[13] OpenAI. OpenAI batch API. `https://platform.openai.com/docs/guides/batch`, 2024.

[14] Bowen Pang, Kai Li, Ruifeng She, and Feifan Wang. A hybrid offline-online scheduling method for large language model inference optimization. *arXiv preprint arXiv:2502.15763*, 2025.

[15] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Teola: Towards end-to-end optimization of LLM-based applications. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 25)*, 2025.

[16] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[17] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[18] Wei Zhang, Zhiyu Wu, Yi Mu, et al. JITServe: SLO-aware LLM serving with imprecise request information. *arXiv preprint arXiv:2504.20068*, 2025.

[19] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.