Orbit: Efficient Agentic Inference using Priority Scheduling

Sami Abuzakuk
EPFL
Lausanne, Switzerland
sami.abuzakuk@epfl.ch

Anne-Marie Kermarrec
EPFL
Lausanne, Switzerland
anne-marie.kermarrec@epfl.ch

Palak
EPFL
Lausanne, Switzerland
palak.palak@epfl.ch

Rafael Pires
EPFL
Lausanne, Switzerland
rafael.pires@epfl.ch

Rishi Sharma
EPFL
Lausanne, Switzerland
rishi.sharma@epfl.ch

Martijn de Vos
EPFL
Lausanne, Switzerland
martijn.devos@epfl.ch

Abstract

Large language models (LLMs) are increasingly deployed as autonomous agents that execute complex tasks through long sequences of reasoning steps and tool calls. Serving these agentic workloads at scale is a growing priority, yet existing LLM inference engines are mostly designed for single-pass, chat-style interactions. Agentic tasks, however, can fail, and under system congestion, policies such as first come first serve (FCFS) allocate GPU resources equally to all tasks regardless of their likelihood of success, letting failing tasks inflate queue waiting times for tasks that will complete correctly. We introduce Orbit, a trajectory-aware scheduler that addresses this by dynamically deprioritizing tasks predicted to fail, shifting resources toward likely-correct tasks. We present two scheduling algorithms. Orbit-step is a lightweight heuristic that assigns priority inversely proportional to the number of steps a task has taken, leveraging the empirical observation that longer-running tasks are more likely to fail. Orbit-judge uses an asynchronous LLM-as-a-judge that reads partial execution trajectories and predicts task correctness online, enabling finer-grained prioritization. We evaluate both variants on the GAIA benchmark using the Magentic-One benchmark and the GPT-OSS-120B model. Compared to vLLM's default FCFS policy, Orbit-step and Orbit-judge show 11.2% and 4.3% decrease in average end-to-end latency, respectively, for correct tasks. These results demonstrate that real-time trajectory observation is a practical and effective signal for resource allocation in agentic serving systems.

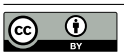
CCS Concepts

• **Computing methodologies** → **Artificial intelligence**.

Keywords

Large Language Models, Agentic Systems, Serving Systems, Scheduling

ACM Reference Format:
Sami Abuzakuk, Anne-Marie Kermarrec, Palak, Rafael Pires, Rishi Sharma, and Martijn de Vos. 2026. Orbit: Efficient Agentic Inference using Priority Scheduling. In *The 6th Workshop on Machine Learning and Systems (EuroMLSys '26), April 27–30, 2026, Edinburgh, Scotland UK*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3805621.3807661



EuroMLSys '26, Edinburgh, Scotland UK

ACM ISBN 979-8-4007-2605-7/26/04
https://doi.org/10.1145/3805621.3807661

1 Introduction

Large language models (LLMs) are rapidly evolving from question-answering systems into autonomous agents that can solve complex user tasks [23]. By augmenting LLMs with capabilities such as external tool invocation, persistent memory, and iterative decision-making, agentic systems enable models to plan, act, and adapt in response to feedback from their environment [20, 24]. These agents dynamically construct execution strategies at runtime, allowing them to carry out complex, multi-step workflows, *e.g.*, coordinating bookings across multiple platforms or managing financial activities on behalf of the user. LLM-based agents are increasingly deployed across diverse application domains, including web browsing [15], software engineering [5], and customer support [22]. Serving these workloads at scale is therefore increasingly critical.

State-of-the-art LLM inference engines such as vLLM [8] are routinely used as backends for agentic systems. However, agentic workloads fundamentally differ from the chat interactions these systems were designed for. Unlike standard single-pass inference, agentic tasks are *multi-step*, *stateful*, and *dynamic*. A single user task unfolds as a *trajectory*: a long sequence of reasoning steps, tool calls, and observations where each step conditions on the outcomes of previous steps [20, 24]. The trajectory length is unpredictable, and its outcome is uncertain until the very end [18]. More critically, *agentic tasks can fail*: suboptimal decisions compound across multiple steps, tool outputs are misinterpreted, and the agent drifts away from a viable execution path. This creates inefficiency under system congestion.

Existing schedulers, such as FCFS in vLLM [8], are well-suited for chat-style workloads where each request translates to a single-pass generation and the success in producing a response is implied. Hence, they allocate GPU resources equally to all tasks regardless of their prospects. This is suboptimal for agentic workloads, as tasks headed toward failure consume the same compute as tasks that will ultimately succeed, inflating queue waiting times and reducing the throughput of correct tasks. Put differently, *correct tasks pay the price for failing ones*, a problem that only worsens as load increases.

To address this, we introduce Orbit, a trajectory-aware scheduling framework for agentic inference that dynamically adjusts request priorities based on real-time signals from a task's execution trajectory. We present two concrete scheduling algorithms under

this framework. Orbit-step is a lightweight heuristic that prioritizes tasks inversely proportional to the number of steps they have taken. The key insight is that *the number of steps a task has taken is a strong signal for predicting correctness.* As tasks progress, they accumulate more opportunities for errors: the context grows, suboptimal decisions compound, tool outputs may be misinterpreted, and the agent may drift further from viable execution paths. By progressively deprioritizing longer-running tasks, Orbit-step ensures GPU resources are allocated preferentially toward tasks more likely to succeed. Orbit-judge complements this approach with a semantic predictor. The sequence of reasoning traces, tool calls, and observations frequently reveals whether an agent is still making meaningful progress or has already gone astray. Orbit-judge therefore employs an *LLM-as-a-judge* that reads the intermediate trajectory and emits a lightweight binary verdict: on-track or off-track. This signal captures early indicators of failure beyond what step count alone can provide. The judge runs asynchronously and off the critical inference path, using the same model and GPU pool as the agentic tasks, thus refining priorities without blocking task execution and requiring no additional models or setup.

We implement both Orbit-step and Orbit-judge, integrate them in the vLLM serving system, and evaluate their effectiveness on the GAIA benchmark [14] using the Magentic-One [3] agentic framework and GPT-OSS-120B [17] as the backbone LLM. We compare Orbit against vLLM with its default FCFS scheduling policy, representing the current state-of-the-art for serving agentic workloads. Our results demonstrate that both algorithms significantly improve the efficiency of serving agentic workloads. Compared to the default policy in vLLM, Orbit-step and Orbit-judge reduce the average end-to-end latency of correct tasks by 11.2% and 4.3%, respectively.

This demonstrates that observing task trajectories in real-time and using them for scheduling decisions provides a practical and effective signal for improving resource allocation for agentic AI workload serving.

Contributions. This paper makes the following contributions:

(1) We introduce Orbit-step and Orbit-judge, two trajectory-aware schedulers that dynamically adjusts task priorities based on the likelihood of success (Section 3).
(2) We evaluate Orbit-step and Orbit-judge on the GAIA benchmark and find that both schedulers reduce end-to-end latency of correct tasks compared to the default FCFS policy in vLLM (Section 4).

2 Background and problem description

We start by providing a brief overview of LLM-based agents and how agentic inference systems operate. We then formulate the problem that this work addresses.

2.1 LLM-based agents

Agentic AI describes systems in which an LLM serves as the core decision-making module, autonomously working toward the user-defined goals through interaction with an external environment. Rather than generating a single response, the agent maintains an evolving internal state, such as contextual information, intermediate results, and stored memory, and selects subsequent actions conditioned on this state. These actions typically involve invoking external tools, which may retrieve information, update persistent data, or trigger effects in downstream services.

A widely adopted paradigm for such systems is the ReAct (Reasoning and Acting) framework [24], which interleaves internal reasoning with external execution. In this setting, the LLM alternates between producing reasoning traces that reflect its current plan and issuing tool calls that interact with the environment. The process begins with a user prompt that the LLM analyzes to formulate an initial plan. Based on its reasoning, the model may invoke a tool, whose output is returned as an observation. This observation is incorporated into the agent's state and informs the next reasoning step. The cycle of reasoning, acting, and observing continues until the agent decides that the objective has been achieved or that no further progress can be made. We refer to this iterative execution pattern as the *agent workflow*. By tightly coupling reasoning with action and feedback, the ReAct loop enables agents to decompose complex objectives into manageable sub-tasks, adapt to new information, and recover from intermediate mistakes during execution.

2.2 Inference in agentic AI workloads

In an agentic system, LLM inference occurs at every reasoning step of the agent workflow. Each time the agent decides how to proceed, whether to invoke a tool, interpret an observation, or synthesize a final answer, it issues a new LLM request to the inference backend. Concretely, in a ReAct-style loop, inference is triggered whenever the agent: *(i)* decides which tool to invoke, *(ii)* interprets tool outputs, or *(iii)* produces the final response. From the serving system's perspective, a single user task is therefore not a single request, but a sequence of interdependent requests issued over time. Each of these stages requires a forward pass through the LLM, typically consisting of a prefill phase followed by autoregressive decoding. Between these inference calls, the system may execute external tools, which introduce latency but do not incur work for the LLM serving system.

Existing LLM serving systems such as vLLM are designed primarily for traditional chat-based workloads, where each user interaction corresponds to a single request-response cycle. In this setting, requests are largely independent, short-lived, and structurally similar. Schedulers therefore optimize for per-request latency and throughput using techniques such as continuous batching and FCFS policies, typically without awareness of the application being executed.

2.3 Problem description

Agentic workloads differ from traditional chat workloads in two key ways. First, requests are not independent: multiple inference calls belong to the same long-running task and form a trajectory of execution. Second, the number of steps required to complete a task is unpredictable, as execution depends on how the interaction unfolds. Furthermore, agentic tasks have a binary notion of correctness: a task either succeeds or fails. From a system perspective, only successful task completions provide value to the user; compute effort spent on tasks that ultimately fail does not contribute to useful throughput.

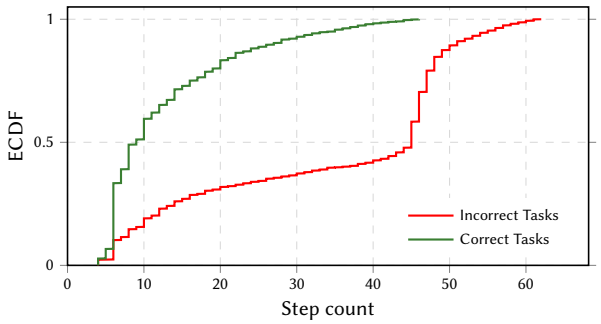


Figure 1: Distribution of steps taken by correct and incorrect tasks in the GAIA benchmark. Tasks that require more steps are more likely to fail.

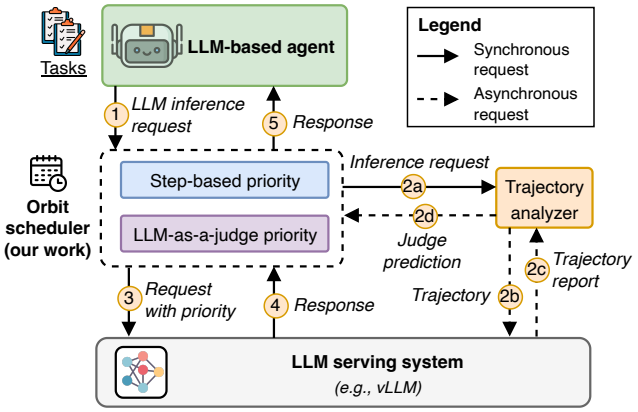


Figure 2: The system architecture and workflow of Orbit-step and Orbit-judge.

Under a light load of the LLM serving system, this distinction is irrelevant since sufficient resources exist to serve all tasks. However, under congestion, GPU resources become a bottleneck. In this regime, treating all inference requests equally can lead to substantial compute being allocated to tasks that are unlikely to succeed, increasing queueing delays for tasks that would otherwise complete correctly. In other words, it is undesirable for tasks that are likely to fail to prolong the completion of tasks that are more likely to succeed.

Rather than minimizing average per-request latency, under system congestion we aim to: minimize the end-to-end latency of correct tasks. To achieve this, the serving system must take the characteristics of running tasks, not just individual requests, and prioritize trajectories that are more likely to lead to successful completion. Therefore, the key technical challenge that this work needs to address is to *predict the correctness of ongoing tasks* in real time and adapting scheduling decisions accordingly.

3 Design of Orbit

Orbit is a priority scheduler to improve the efficiency of agentic inference workloads. Our key design goal is to increase the throughput and reduce average end-to-end latency of correct tasks by observing task progression in real-time and schedule associated inference requests accordingly.

3.1 Predicting task success

Orbit requires some indicators that can predict whether a task is going to fail or succeed. We empirically found that the number of steps and the characteristics of the trajectory are robust indicators that can be leveraged to make this prediction. We motivate the insight behind these signals next.

Step count. As agentic tasks progress through more steps, they accumulate more opportunities for errors: suboptimal decisions compound, tool outputs may be misinterpreted, and tasks drift from viable paths. Importantly, each additional step also increases the context length of the agent. The prompt grows with prior reasoning traces, tool calls, and observations, making the model condition on an increasingly long and noisy history. As context windows grow, relevant signals become diluted among irrelevant or erroneous intermediate states, increasing the likelihood of hallucinations or inconsistent reasoning [10]. This effect is particularly pronounced for models with smaller context capacities, where later steps may exceed the model's ability to reliably attend to earlier information.

We empirically show this phenomenon in Figure 1, which shows the distribution of the number of steps for correct and incorrect tasks in the GAIA benchmark. We run tasks in this benchmark four times and record if a task failed or succeeded. Incorrect tasks take more steps than correct tasks, with clear separation between distributions. Specifically, 50% of the correct tasks take 10 steps to complete on average, compared to 45 steps for incorrect tasks. The inability of agents to deal with long-running tasks is also reported in related works and observed for different benchmarks [13, 21, 27]. Thus, a scheduler can leverage this signal to deprioritize tasks that are unlikely to succeed.

Trajectories. A trajectory can provide early indicators of potential failure. Inappropriate tool calls or flawed reasoning steps often signal that the current task progression will lead to an erroneous outcome. For example, if an agent invokes a summarization API during a data extraction task, this mismatch reveals both a misunderstanding of the task intent and degraded reasoning quality. Therefore, our idea is to leverage the capabilities of an LLM to predict whether the task is making meaningful progression towards its objective. This approach, commonly referred to as *LLM-as-a-judge* [26], uses a language model to evaluate the quality of another model's outputs. Concretely, we prompt the judge model with the task formulation and the trajectory produced so far, and ask it to output a binary progress signal that indicates whether the current behavior is on track toward successful completion or shows signs of incorrect reasoning or tool use. The trajectory includes all reasoning steps and tool calls from the agents, presented in the order of their invocation. Since Orbit does not modify the tasks and only reorders their execution based on predicted success, the overall task success rate remains unchanged.

3.2 System design

Based on the above insights, we design two Orbit schedulers: Orbit-step that is based on the current task step count, and Orbit-judge that leverages LLM-as-a-judge to analyze the current trajectory and predict failure.

3.2.1 System model. We model the system as a set of concurrently executing tasks. Each task $\mathcal{T}$ consists of a sequence of agent steps, where each step produces one inference request to the LLM serving system. An inference request I is represented as a tuple $I = (id, s, t)$, where $id \in \mathbb{N}$ is a unique task identifier, $s \in \mathbb{N}$ is the number of steps the task has completed so far, and t is the partial trajectory: the concatenation of all reasoning steps, tool calls, and observations produced by the task up to step s.

3.2.2 Design of Orbit-step. Figure 2 shows the workflow of both Orbit-step and Orbit-judge. The Orbit scheduler is implemented as a layer between the LLM-based agent and LLM serving system and intercepts LLM inference requests and responses. We assume that the LLM-based agent is currently executing one or more tasks. When an LLM-based agent completes a step and sends an inference request to the serving system ①, the Orbit scheduler intercepts it. We next describe the actions taken in Orbit-step and Orbit-judge.

When Orbit-step receives the inference request, it assigns a priority score to the request that is inversely proportional to s. Given step s, we compute the priority as follows:

$$\text{STEP\_PRIORITY}(s) = \left\lfloor \frac{s}{10} \right\rfloor \tag{1}$$

Step counts are grouped into buckets of ten: steps 0–9 map to priority 0, steps 10–19 to priority 1, and so on. Tasks in the first bucket receive priority 0 (highest), as they are statistically more likely to complete correctly (Figure 1). Tasks in higher buckets are progressively pushed back while still making forward progress when resources are available. We then forward the prioritized request to the LLM serving system ③. When Orbit receives the response ④, it forwards the response back to the LLM agent ⑤.

3.2.3 Design of Orbit-judge. Orbit-judge introduces a trajectory analyzer that uses a judge LLM to predict, based on the current trajectory t whether a task is going to fail or succeed. For each task, we maintain a sequence of judge predictions $P = (P_1, \ldots, P_n)$ where $P_i \in \{fail, success\}$ is the judge's binary prediction at step i and n is the number of predictions made so far. We use array-style indexing for recent predictions, where $P[-1]$ denotes the most recent prediction, $P[-2]$ the second most recent, and so on. Orbit-judge assigns a discrete priority level $p \in high, medium, low, lowest$ based on the recent failure history. This works as follows: if the cumulative number of failure predictions exceeds a threshold K_{fail}, the task is assigned the lowest possible priority. If the two most recent predictions are both `fail`, the task is likely drifting and receives `LOW` priority. If only the most recent prediction is `fail`, the task receives `MEDIUM` priority. If the most recent prediction is `correct`, the task receives `HIGH` priority. Intuitively, repeated failure signals indicate that the task is likely off-track, and we progressively deprioritize it. We define F to be the total number of failure predictions so far. The priority assignment function is then defined as:

$$p(P) = \begin{cases} \texttt{LOWEST}, & \text{if } F > K_{\text{fail}}, \\ \texttt{LOW}, & \text{if } P[-1] = \texttt{fail} \wedge P[-2] = \texttt{fail}, \\ \texttt{MEDIUM}, & \text{if } P[-1] = \texttt{fail}, \\ \texttt{HIGH}, & \text{if } P[-1] = \texttt{success}. \end{cases}$$

Orbit-judge forwards the inference request to the trajectory analyzer which stores for each task its prediction history P ②a. This analyzer then creates a separate inference request, based on attached trajectory t ②b and a system prompt (see Appendix B and C). The analyzer receives the trajectory report from the serving system ②c, sends back the prediction to Orbit ②d who assigns the appropriate priority based on the function p defined above. This prioritized request is sent to the serving system ③. The serving system processes the request according to its computed priority and returns the generated response ④, which Orbit passes back to the agent ⑤. Because trajectory analysis runs asynchronously, it never blocks the critical path: requests that arrive before a judge result is ready are scheduled using the step-based priority alone.

In both the systems, Orbit-step and Orbit-judge, tasks with equal priority are tie-broken using the expected decode length of the requesting agent, prioritizing requests from subordinate agents over the orchestrator. This is motivated by our offline analysis, which shows that the orchestrator typically has longer decode lengths due to responsibilities such as high-level planning. Further details are in Table 2 in Appendix A.

4 Evaluation

We implement Orbit as a scheduling middleware that intercepts LLM requests between the agentic system and the inference backend. Our evaluation addresses three questions:

(1) What is the performance of Orbit-step and Orbit-judge in terms of average end-to-end latency of tasks, compared to the default FCFS policy in vLLM (Section 4.2)?
(2) How effective is the LLM-as-a-judge mechanism in Orbit-judge at identifying failing tasks (Section 4.3)?
(3) What is the overhead of Orbit-step and Orbit-judge in terms of additional queuing delays (Section 4.4)?

4.1 Experimental setup

We next describe our implementation and experimental setup.

4.1.1 Implementation. Both Orbit-step and Orbit-judge are implemented as a lightweight wrapper around vLLM's scheduler and replace the default FCFS policy with priority-aware scheduling. vLLM is a state-of-the-art LLM serving system that integrates continuous batching and PagedAttention to reduce KV cache fragmentation. Orbit maintains a *task registry*, a shared in-memory map from task UUID to execution state (step count, partial trajectory, and latest judge result), updated on every request arrival and judge callback. Priority values are real-valued scalars where a lower number indicates higher scheduling precedence.

4.1.2 Agentic System. We use Magentic-One as our agentic framework. A detailed description can be found in Appendix A.

4.1.3 LLM and hardware. We use GPT-OSS-120B as the backbone LLM, which is a state-of-the-art mixture-of-experts (MoE) model with 117×10^9 parameters. The model activates 5.1×10^9 parameters per token and can run efficiently on a single 80 GB GPU. Its optimization for high-reasoning, agentic workflows, and tool use makes it a suitable backbone for our experimental setup and in the context of our work. We serve GPT-OSS-120B using vLLM as the underlying inference engine on 4 NVIDIA A100 GPUs (80 GB HBM2e each). We use the same model to perform trajectory analysis in Orbit-judge.

4.1.4 Workload. We evaluate our scheduler on the GAIA benchmark, which tests AI assistants on multi-step, tool-augmented, real-world tasks (e.g., web browsing, code execution, and file processing). The benchmark provides an automated evaluation by matching final outputs against expected answers. We use the GAIA validation Level 1 Magentic-One which contains 53 tasks. A task is marked as failed if the final output does not match the expected answer or if execution terminates upon reaching the maximum round limit. We also observe cases where the agent produces the correct final answer despite a sequence of incorrect intermediate steps when the task terminates due to the round limit. As these outputs are not derived from a consistent or verifiable reasoning process, but instead reflect memorized responses, we conservatively classify such cases as incorrect.

In each experiment run, we submit a total of 212 tasks (53 unique tasks repeated 4 times each) at a Poisson arrival rate of $\lambda = 0.5$ tasks/second to simulate realistic concurrent agentic workloads and saturate our system. We fix the max batch size across all experiments to 16. Furthermore, for Orbit-judge we set $K_{fail} = 10$ since we empirically found that this number of consecutive negative predictions is an effective predictor for task failure.

The results are averaged over five runs for each scheduler.

4.1.5 Baseline. We compare Orbit against the default FCFS policy in vLLM, which is application-unaware and treats each request as an independent unit of work, without any consideration of the task's execution state or likelihood of success. This represents the current state-of-the-art for serving agentic workloads.

4.2 Performance of Orbit

We first quantify the performance of all, correct and incorrect tasks in Orbit and our baseline, in terms of average end-to-end latency of tasks. The average end-to-end latency of tasks is the average time from task submission to completion. Table 1 summarizes the overall performance of Orbit compared to the FCFS baseline. On average, a task completes in 1704 seconds and incorrect tasks take significantly more time: 2440 seconds on average. We attribute this to the fact that incorrect tasks tend to run for a higher number of steps (also see Figure 1). Orbit-step and Orbit-judge reduce the average end-to-end latency of correct tasks by 11.2% and 4.3%, respectively. These results show that both Orbit schedulers are effective at prioritizing correct tasks. This comes, however, with an increase in average task latency of incorrect tasks: 8% for Orbit-step and 10% for Orbit-judge. Figure 3 shows the end-to-end latency distribution of correct tasks for Orbit and FCFS.

Table 1: Average end to end latency for correct and incorrect tasks with FCFS, Orbit-step and Orbit-judge.

Scheduler	Correct [s.]	Incorrect [s.]	All [s.]
FCFS	901.2	2439.5	1703.5
Orbit-step	800.6 (-11.2%)	2635.1 (+8.0%)	1750.6 (+2.7%)
Orbit-judge	862.4 (-4.3%)	2686.9 (+10.1%)	1837.8 (+7.7%)

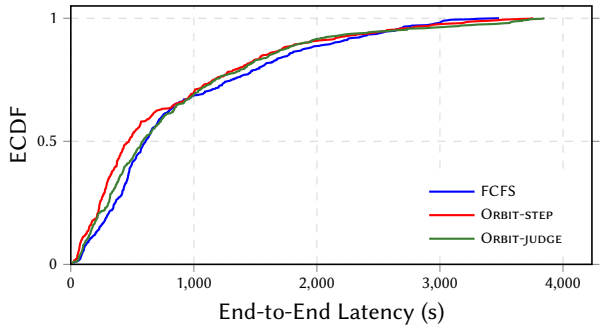


Figure 3: The distribution of end-to-end latencies for correct tasks on the GAIA benchmark, for both Orbit-step and Orbit-judge, and the FCFS default policy in vLLM.

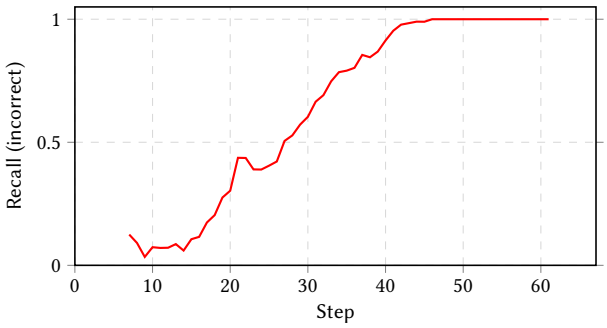


Figure 4: Recall of incorrect tasks (incorrect as positive class) as a function of step. Recall increases as tasks progress, reflecting the Orbit-judge's growing ability to identify failing tasks later in their trajectory.

4.3 Performance of the Orbit-judge predictor

We utilize LLM-as-a-judge in Orbit-judge to identify tasks that will potentially fail. To demonstrate the performance of this classifier, we present the recall per step when classifying incorrect tasks in Figure 4. We observe a low recall at the early stages in agentic workflows as the LLM-as-a-judge classifies all tasks as correct, *i.e.*, there are almost no true positives. This is because there is not enough semantic information in the trajectory data to classify tasks as incorrect at this stage. Between steps 15 and 40, the contextual information becomes relevant for the LLM-as-a-judge to discriminate between correct and incorrect tasks. We see this in the near-linear increase in recall as the number of true positives increases with increasing steps. Finally, for a step count beyond 45, the task is bound to fail because of the large context length

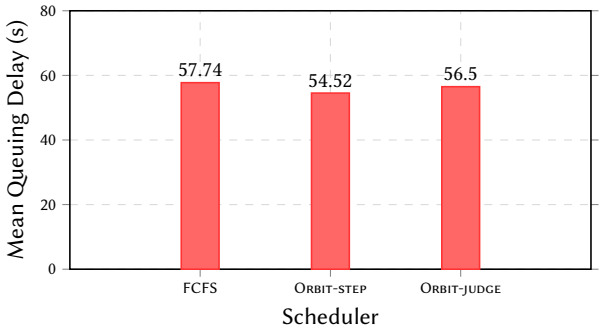


Figure 5: Average request queuing delay for FCFS, Orbit-step and Orbit-judge at batch size 16.

and accumulated mistakes, and the classifier predicts it perfectly. We believe the LLM-as-a-judge mechanism in Orbit-judge is the most useful in the middle range of step count, where the semantic information in the agentic workflow starts to become useful.

4.4 Overhead analysis

Figure 5 shows the average queuing delay for FCFS, Orbit-step, and Orbit-judge at the request level. Orbit-step reduces the average queuing delay by 5%, while Orbit-judge decreases it by 2%. Orbit-step and Orbit-judge prioritizes correct tasks, which complete earlier and progressively reduce the number of in-flight requests and, in turn, the queuing delay.

4.5 Discussion

Our results demonstrate that trajectory-aware scheduling improves the efficiency of correct tasks in serving agentic inference workloads. By observing task execution in real-time and dynamically adjusting priorities, Orbit achieves improvements in average end-to-end latency of correct tasks compared to application-unaware scheduling policies. Even though Orbit shows promising results, there is still room for improvement, particularly in developing more sophisticated predictors that can identify failing tasks earlier in their execution.

Orbit-judge performance is sensitive to the quality of signals available in the task trajectory. Currently, the judge relies primarily on step counts, tool calls and orchestrator self-assessments; richer signals could improve prediction accuracy. As a result, Orbit-judge shows lower improvement over FCFS than Orbit-step, highlighting the need for richer trajectory signals beyond step counts to improve prediction accuracy. Beyond improving prediction quality, future work could leverage high-confidence failure predictions to proactively terminate tasks early. While Orbit currently reorders execution without reducing total compute, early termination could free GPU resources and further improve throughput under congestion.

An interesting direction for future work is to combine Orbit-step and Orbit-judge into a unified scheduler that jointly reasons about trajectory length and recent correctness signals. While these mechanisms are complementary in principle, their interaction may introduce non-trivial interactions and require careful design.

5 Related work

LLM Serving Systems. Systems such as vLLM [8], Orca [25], and TensorRT-LLM [16] optimize throughput and latency for standard inference through continuous batching, KV cache, and request scheduling. These systems target each request as independent and optimize individual LLM calls. However, such LLM serving systems perform suboptimally on agentic tasks as they lack mechanisms to observe multi-step execution. Orbit, on the other hand, targets agentic workflows through its novel trajectory-aware scheduling.

Request Scheduling in LLMs. Recent work has explored scheduling policies in LLM systems. Sarathi [1] proposes chunked-prefill, ALTO [19] proposes prompt-aware scheduling, and DistServe [28] disaggregates the prefill and decode stages. These approaches still treat requests independently. Autellix [12] utilizes program-level context in agentic workloads to minimize waiting times of requests. Orbit is complementary to these approaches. We focus on multi-step trajectories in agentic workflows, predict task correctness, and dynamically deprioritize tasks unlikely to succeed.

Agentic Frameworks. Frameworks like Magentic-One [3], AutoGPT [2], LangChain [9], and ReAct [24] enable multi-step reasoning and tool use, demonstrating success on benchmarks like GAIA [14] and WebArena [29]. While these frameworks improve agent capabilities, they do not address efficient serving at scale. Orbit is complementary, focusing on serving infrastructure that optimizes throughput and latency of correct task completions.

LLM-as-a-Judge. LLMs have been used as substitutes for human evaluators to judge LLM outputs [11, 26]. Many works also extend the paradigm by fine-tuning models to improve evaluations [4, 6, 7]. These works, however, use LLMs as offline evaluators. In contrast, Orbit employs an LLM online during the agentic run to reduce the latency of correct tasks with minimal overheads. Furthermore, Orbit does not require any fine-tuning of foundational models.

6 Conclusion

We presented Orbit, a trajectory-aware scheduler that prioritizes tasks based on predicted correctness. We introduced two variants: Orbit-step, a lightweight step-count heuristic, and Orbit-judge, which augments scheduling with an asynchronous LLM-as-a-judge for finer-grained online predictions. Integrated into vLLM and evaluated on the GAIA benchmark with Magentic-One and GPT-OSS-120B, Orbit-step and Orbit-judge reduce average end-to-end latency by 11.2% and 4.3%, respectively, compared to the default FCFS policy in vLLM. Our results show that execution trajectories provide a practical and powerful signal for scheduling agentic inference. As LLM systems increasingly operate as long-running agents, trajectory-aware serving will be essential for efficient and scalable deployment.

7 Acknowledgments

This work has been funded by the Swiss National Science Foundation, under the project "FRIDAY: Frugal, PrivacyAware and Practical Decentralized Learning", SNSF proposal No. 10.001.796.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming

throughput-latency tradeoff in llm inference with sarathi-serve. pages 117–134, 2024.

[2] autogpt. autogpt. https://github.com/Significant-Gravitas/AutoGPT, 2023. Accessed: 2026-02-02.

[3] Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, et al. Magentic-one: A generalist multi-agent system for solving complex tasks. *arXiv preprint arXiv:2411.04468*, 2024.

[4] Seungju Han, Kavel Rao, Allyson Ettinger, Liwei Jiang, Bill Yuchen Lin, Nathan Lambert, Yejin Choi, and Nouha Dziri. Wildguard: Open one-stop moderation tools for safety risks, jailbreaks, and refusals of llms. *Advances in neural information processing systems*, 37:8093–8131, 2024.

[5] Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30, 2025.

[6] Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, et al. Prometheus: Inducing fine-grained evaluation capability in language models. In *The Twelfth International Conference on Learning Representations*, 2023.

[7] Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. Prometheus 2: An open source language model specialized in evaluating other language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 4334–4353, 2024.

[8] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph 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*, pages 611–626, 2023.

[9] langchain. Langchain. https://github.com/langchain-ai/langchain, 2023. Accessed: 2026-02-02.

[10] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

[11] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 2511–2522, 2023.

[12] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. Autellix: An efficient serving engine for llm agents as general programs. *arXiv preprint arXiv:2502.13965*, 2025.

[13] Elliot Meyerson, Giuseppe Paolo, Roberto Dailey, Hormoz Shahrzad, Olivier Francon, Conor F Hayes, Xin Qiu, Babak Hodjat, and Risto Miikkulainen. Solving a million-step llm task with zero errors. *arXiv preprint arXiv:2511.09030*, 2025.

[14] Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

[15] Liangbo Ning, Ziran Liang, Zhuohang Jiang, Haohao Qu, Yujuan Ding, Wenqi Fan, Xiao-yong Wei, Shanru Lin, Hui Liu, Philip S Yu, et al. A survey of webagents: Towards next-generation ai agents for web automation with large foundation models. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 6140–6150, 2025.

[16] NVIDIA. TensorRT-LLM: A tensorrt toolbox for optimized large language model inference. https://github.com/NVIDIA/TensorRT-LLM, 2023. Accessed: 2026-02-02.

[17] OpenAI. Gpt-oss-120b. https://huggingface.co/openai/gpt-oss-120b, 2025. Open-weight 120B parameter mixture-of-experts model. Accessed: 2026-02-24.

[18] Melissa Z Pan, Mert Cemri, Lakshya A Agrawal, Shuyi Yang, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Kannan Ramchandran, Dan Klein, et al. Why do multiagent systems fail? In *ICLR 2025 Workshop on Building Trust in Language Models and Applications*, 2025.

[19] Keshav Santhanam, Deepti Raghavan, Muhammad Shahir Rahman, Thejas Venkatesh, Neha Kunjal, Pratiksha Thaker, Philip Levis, and Matei Zaharia. Alto: An efficient network orchestrator for compound ai systems. In *Proceedings of the 4th Workshop on Machine Learning and Systems*, pages 117–125, 2024.

[20] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

[21] Akshit Sinha, Arvindh Arun, Shashwat Goel, Steffen Staab, and Jonas Geiping. The illusion of diminishing returns: Measuring long horizon execution in llms. *arXiv preprint arXiv:2509.09677*, 2025.

[22] Haoxin Wang, Xianhan Peng, Huang Cheng, Yizhe Huang, Ming Gong, Chenghan Yang, Yang Liu, and Jiang Lin. Ecom-bench: Can llm agent resolve real-world e-commerce customer support issues? In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 276–284, 2025.

[23] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

[24] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

[25] 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.

[26] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

[27] Wenhao Zheng, Xinyu Ye, Peng Xia, Fang Wu, Linjie Li, Weitong Zhang, Lijuan Wang, Yejin Choi, Yun Li, and Huaxiu Yao. The agent's marathon: Probing the limits of endurance in long-horizon tasks.

[28] 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)*, pages 193–210, 2024.

[29] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

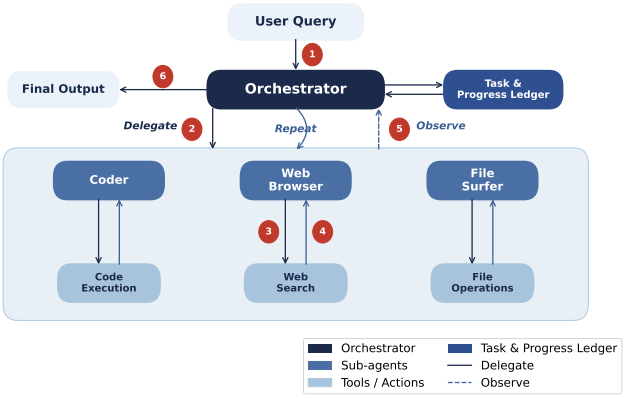


Figure 6: Architecture of Magentic-One. The Orchestrator receives a user query, maintains a task and progress ledger, and delegates execution to one sub-agent at a time. Each sub-agent interacts with its corresponding tool and reports observations back to the Orchestrator, which uses them to decide the next action. Once the task is complete, the Orchestrator produces the final output.

Table 2: Average decode length per agent

Agent	Avg. Decode Length
Orchestrator	11,619.65
Web Surfer	2,766.17
Coder	956.31
File Surfer	619.48

A Magentic One

MAGENTIC-ONE implements a multi-agent architecture using AUTOGEN where an orchestrator agent coordinates a set of specialized sub-agents, each responsible for specific tasks such as code execution, web browsing, and file manipulation. The orchestrator decomposes the user's query into sub-tasks, delegates them to the appropriate agents, and synthesizes their outputs to produce a final response. Figure 6 illustrates this architecture.

B System Prompt

Figure 7 shows the system prompt used by the judge in ORBIT-JUDGE.

C Trajectory for LLM-as-Judge

Figure 8 shows a representative input trajectory fed to ORBIT-JUDGE for prediction. The trajectory spans 4 rounds and 9 steps.

Figure 7: The system prompt used by the judge in Orbit-judge

Judge System Prompt

```
You are an expert at analyzing AI agent tool call trajectories.
BE CONCISE. THINK BRIEFLY. YOUR ENTIRE REASONING PROCESS SHOULD TAKE NO MORE THAN A FEW SENTENCES.

The agent you are analyzing is part of MagenticOne, a multi-agent system built on top of an LLM orchestrator.
MagenticOne consists of the following specialized agents that the orchestrator can delegate to:

  - Orchestrator: The lead agent that plans, delegates tasks to other agents, and synthesizes results.
  - WebSurfer: Browses the web, searches for information, and navigates web pages.
  - FileSurfer: Reads and navigates local files (PDFs, text files, spreadsheets, etc.).
  - Coder: Writes and reasons about code to solve problems programmatically.
  - ComputerTerminal: Executes code and shell commands in a sandboxed environment.

Important: Even if early steps show WebSurfer or FileSurfer struggling, the orchestrator can fall back to Coder +
ComputerTerminal to solve tasks programmatically. Do not penalize the agent prematurely for early failures if there are still
unexplored avenues available.

The system has a maximum of {TOTAL_ROUND} rounds. The trajectory includes [Round X/{TOTAL_ROUND}] markers showing the
orchestrator's self-assessment at each round boundary. Pay close attention to rounds remaining.

You will be given:
  1. The original task/prompt.
  2. The trajectory so far: tool calls with agent reasoning and round boundary assessments.
  3. How many steps and rounds have elapsed.

Based on the task and the trajectory, predict whether the agent will ultimately succeed or fail.

Consider factors like:
  - Is the agent's approach appropriate for the given task?
  - Is the agent making progress toward answering the task or going in circles?
  - Are the tool calls logical and building toward a solution?
  - Are there signs of confusion, repetition, or errors that persist across multiple agents?
  - Is the most recent tool result returning useful information or errors/empty results?
  - Is the agent interpreting tool results correctly and adjusting its strategy?
  - How many rounds remain (out of 20)? Is there realistically enough time to finish?
  - What does the orchestrator's own self-assessment say about progress and loops?

Respond with EXACTLY one JSON object (no markdown, no explanation):
  {{
    "prediction": "Correct" | "Incorrect",
    "confidence": <0.0-1.0>,
    "reasoning": "<one line reason, BE VERY CONCISE>",
  }}

Where:
  - "Correct" means the task will be completed successfully.
  - "Incorrect" means the task will fail. If you believe the failure is due to running out of rounds (the agent was still
making meaningful progress but will hit the 20-round limit), you MUST include the exact token "max_round_reached" in your
reasoning field so this can be tracked separately.
```

Figure 8: A sample input trajectory to ORBIT-JUDGE for prediction.

Task Prompt

How many at-bats did the Yankee with the most walks in the 1977 regular season have that same season?

Step 1 / Round 1 — Orchestrator (reasoning)

Given or verified facts
- The request concerns the New York Yankees, 1977 regular season.
- Statistic of interest: most walks (BB) by any Yankee player.
- Question asks for that player's at-bats (AB) the same season.

Facts to look up
- Player on the 1977 Yankees roster with the highest walk total.
- That player's total BB and AB for the 1977 regular season.
- Sources: Baseball-Reference.com, Retrosheet.org, Baseball Almanac.

Facts to derive
- Identify the walk leader from retrieved data.
- Extract that player's AB from the same batting table.
- Verify figures are regular-season only (exclude postseason).

Educated guesses
- Likely candidates: Reggie Jackson, Thurman Munson, Graig Nettles.
- Typical AB for a full-time 1977 starter: 500–650.

Step 2 / Round 1 — Orchestrator (planning)

Step A – Use WebSurfer to search "1977 New York Yankees team batting stats" on Baseball-Reference; locate BB column; record highest-BB player.

Step B – On the same page, note the AB figure for that player (1977 regular season only).

Step C (optional) – Cross-check on Retrosheet or Baseball Almanac.

Step D – Compile final answer: "The Yankee with the most walks in 1977 was [Player] with [BB] walks and [AB] at-bats."

Agents: WebSurfer (primary); ComputerTerminal (optional, table parsing).

[Round 1/20] — Progress Ledger (Step 3)

```
is_request_satisfied : false
  reason : Walk leader and AB total not yet identified.
is_in_loop : false
is_progress_being_made : false
  reason : Only a plan outlined; no data gathered yet.
next_speaker : WebSurfer
```

Step 4 / Round 2 — WebSurfer (tool: web_search)

query : "1977 New York Yankees team batting statistics Baseball-Reference walks BB"

reasoning : Search Baseball-Reference for the 1977 Yankees batting page to locate the BB column and identify the player with the highest walk total.

tool_output : Search results page returned.

[Round 2/20] — Progress Ledger (Step 5)

```
is_request_satisfied : false
  reason : Search results retrieved; walk leader and AB not yet identified.
is_in_loop : false
is_progress_being_made : true
  reason : Located search results for 1977 Yankees batting stats.
next_speaker : WebSurfer
```

Step 6 / Round 2 — WebSurfer (tool: click)

```
target_id : 83 (Baseball-Reference 1977 New York Yankees page)
reasoning : Open the Baseball-Reference team page to access the full batting table containing BB and AB columns for every player.
tool_output : 1977 Yankees team page loaded successfully.
```

[Round 3/20] — Progress Ledger (Step 7)

```
is_request_satisfied : false
  reason : Correct page opened; walk leader and AB total not yet extracted.
is_in_loop : false
is_progress_being_made : true
  reason : Navigated to correct page; next step is to extract statistics.
next_speaker : WebSurfer
```

Step 8 / Round 3 — WebSurfer (tool: scroll_down)

```
reasoning : Scroll down to reach the Team Batting table on the Baseball-Reference page where per-player BB and AB totals are listed.
tool_output : Team batting table visible. Walk leader identified: Cliff Johnson. AB total extracted from table.
```

[Round 4/20] — Progress Ledger (Step 9)

```
is_request_satisfied : true
  reason : Cliff Johnson identified as walk leader; AB total for the 1977 regular season extracted from Baseball-Reference.
is_in_loop : false
is_progress_being_made : true
  reason : All required data obtained (player name, walk total, AB total).
next_speaker : Assistant
```

ORBIT-JUDGE Output

```
{{
  "prediction" : "Correct",
  "confidence" : 0.95,
  "reasoning" : "Agent navigated directly to Baseball-Reference, located batting
       table in 4 rounds, identified Cliff Johnson as walk leader, and extracted
       AB total without loops or errors."
}}
```