SPECSYS: SELECTIVE SPECULATIVE DECODING AND COMPLETION-AWARE SCHEDULING FOR EFFICIENT AGENTIC LLM INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-powered agents often invoke multiple LLM requests, making LLM inference a major contributor to end-to-end task completion time. Our measurements reveal that the dominant bottleneck shifts with serving load: from decoding at low concurrency to queueing at high concurrency. Speculative decoding (SD) is a state-of-the-art technique for accelerating LLM inference, but its benefits diminish as serving load increases and can even become negative under high load. Existing adaptive SD methods cannot eliminate this problem because they lack native non-SD execution, while conventional serving schedulers do not fully exploit workflow structure and completion progress of agent tasks. To address these challenges, we design SPECSYS, an efficient inference system for agent workloads that enables load-aware selective SD with completion-aware scheduling. SPECSYS exposes an externally controllable runtime interface that enables complete iteration-level switching between native SD and non-SD execution. It also employs a lightweight history-based estimator to predict the remaining steps of each agent task based on its workflow state and request characteristics. SPECSYS adapts these mechanisms with serving load: it selectively enables SD to reduce decoding latency under low load, while prioritizing tasks with fewer predicted remaining steps under high load to reduce queueing delay. Across three LLMs, four agent workloads, three agent workflows, and two hardware platforms, SPECSYS consistently reduces average job completion time (JCT) over strong baselines, including ThunderAgent, vLLM Dynamic SD, and SGLang Adaptive SD, achieving up to a 2.10× speedup under resource-constrained settings. The source code of SPECSYS is anonymously available at `https://anonymous.4open.science/r/SpecSys-Code-E03D/`.

1 INTRODUCTION

Large language models (LLMs) have evolved from chatbots to autonomous agents capable of exploring and solving complex tasks, such as programming (Austin et al., 2021), web search (Nakano et al., 2021) and environment interaction (Yang et al., 2023). Representative agent workflows, including ReAct (Yao et al., 2023), Plan-and-Solve (Wang et al., 2023), Reflexion (Shinn et al., 2023) and AgentFlow (Li et al., 2026b), typically execute an agentic request through multiple interdependent stages and LLM calls. As a result, users care about the end-to-end completion time of the entire workflow, which is often dominated by LLM generation, posing significant efficiency challenges for existing LLM inference systems. Our measurements (Appendix A) reveal that the dominant latency bottleneck of agentic LLM requests varies with concurrency. Under low concurrency, decode time dominates request latency, whereas queueing delay grows rapidly with increasing concurrency and eventually becomes the dominant latency component.

Speculative decoding (SD) is a widely adopted technique to accelerate LLM decoding by generating candidate tokens with a lightweight drafting mechanism and verifying them in parallel with the target model (Leviathan et al., 2023; Chen et al., 2023; Li et al., 2024). Agent workloads provide additional opportunities for SD due to their repetitive and structured generation patterns, such as recurring intermediate outputs, tool calls, and schema-constrained responses (Oliaro et al., 2025; Xia et al., 2026; Wang et al., 2026). However, the effectiveness of SD is highly sensitive to batch size.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

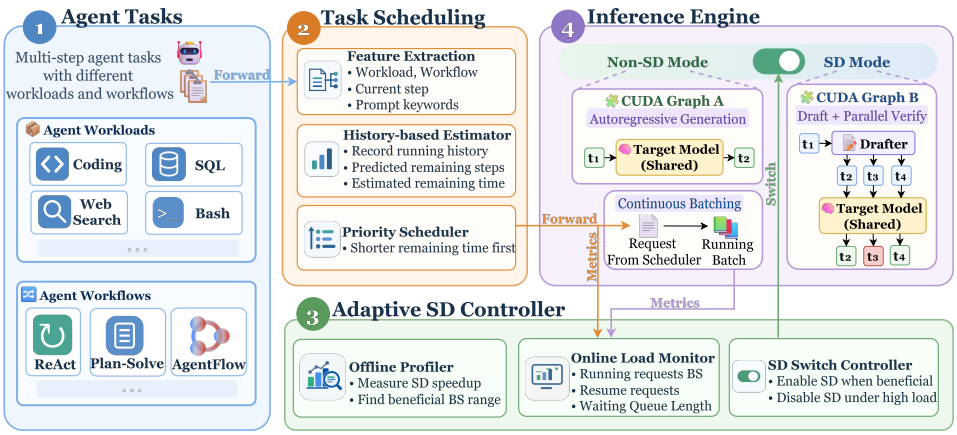

Figure 1: Overview of SpecSys. SpecSys combines remaining-step-aware task scheduling with adaptive SD control. The scheduler estimates the remaining steps of agent jobs and prioritizes shorter jobs, while the SD controller selects between native SD and non-SD execution based on offline profiling and online serving load.

Previous studies have shown that SD speedup for LLM inference decreases as batch size increases and can eventually turn into performance degradation (Liu et al., 2026b; Marzollo et al., 2026; Liu et al., 2024). Our measurements (Appendix B) reveal the same trend for agent workloads. In online agent serving, however, the running batch size varies substantially over time due to dynamic request arrivals and intermittent tool execution between LLM calls.

Recent dynamic SD approaches adapt the draft length according to runtime signals such as batch size and token acceptance rate, and may reduce the draft length to zero when speculation is unfavorable (Liu et al., 2024; Huang et al., 2025a; Li et al., 2026a). But these approaches perform adaptation internally within the SD framework rather than explicitly switching between native SD and non-SD execution paths. Our experiments show that such internal adaptation can still incur noticeable overhead compared with native non-SD execution (Appendix C). Moreover, their systems typically rely on predefined switching policies, without exposing runtime control to external schedulers, which limits customized SD selection based on higher-level scheduling decisions.

These observations raise two key challenges:

1. **How to cleanly switch between native SD and non-SD modes based on external decisions with lower runtime overhead?**
2. **How to reduce end-to-end latency under varying concurrency with shifting bottlenecks?**

To address these challenges, we design SpecSys, an agent-oriented LLM inference system comprising four components, as shown in Figure 1. **Agent Tasks** represent multi-step agent tasks across diverse workloads and workflows and forward their LLM calls to the scheduler. **Task Scheduling** extracts task-level features, estimates remaining execution steps from execution history, and prioritizes pending tasks. The **Adaptive SD Controller** combines offline profiling with online serving metrics to determine the inference mode at runtime. Finally, the **Inference Engine** executes requests through native non-SD or SD paths under continuous batching, reporting runtime metrics and applying controller-issued mode-switches.

To realize SpecSys, we make the following technical contributions:

Externally Controllable SD Switching. We extend the inference engine with an externally controllable switching interface that enables complete iteration-level switching between native SD and non-SD execution modes, reducing runtime overhead by up to 71.3% (Appendix C).

Lightweight Agent Remaining-Step Prediction. We design a lightweight history-based estimator that predicts an agent job's remaining steps from its workload, workflow, current step, and prompt-derived keywords, achieving 84.1% average pair prediction accuracy and 82.2% top-10% recall.

Load-Adaptive Agent Scheduling and SD Selection. We selectively enable SD based on offline profiling under low concurrency, while disabling SD and prioritizing jobs with fewer predicted remaining steps under high concurrency.

Extensive evaluations show that SPECSYS improves agent serving efficiency across diverse settings, with up to a 2.10× speedup in average JCT under resource-constrained environments.

2 RELATED WORK

2.1 ADAPTIVE SPECULATIVE DECODING

Existing token-level adaptive SD approaches can be broadly divided into two categories: one line of work dynamically adjusts drafting parameters, particularly the draft length, based on signals such as token acceptance rate, serving load, batch size, SLOs, and runtime performance (Huang et al., 2025a; Liu et al., 2024; Hou et al., 2025; Huang et al., 2025b; Li et al., 2026a; Zhang et al., 2026a), with similar mechanisms also adopted by vLLM's Dynamic SD (Kwon et al., 2023) and SGLang's Adaptive SD (Zheng et al., 2024). Another line of work instead focuses on verification optimization, dynamically allocating or pruning the verification workload across requests according to SLOs, token acceptance rate, and system load to improve batched verification efficiency (Li et al., 2026c; Liu et al., 2026a; Hu et al., 2026a; Cheng et al., 2026). These approaches optimize SD internally, typically using a zero draft length as a pseudo-non-SD configuration with residual overhead and fixed adaptation logic. In contrast, SPECSYS decides whether to enable SD, switches directly between native SD and non-SD paths, and exposes a runtime interface for arbitrary upper-layer control.

Beyond conventional token-level SD, recent work extends speculation to semantic-level reasoning (Pan et al., 2025; Hu et al., 2026b), and further to agent planning, action, or tool execution (Hua et al., 2025; Guan et al., 2026; Ye et al., 2026). These approaches redesign the upper-level agent workflow, whereas SPECSYS preserves the workflow and applies SD to LLM inference iterations within each agent task, making the two directions complementary. In principle, our remaining-step prediction method could also be extended to such emerging speculative agent workflows by scaling the predicted number of remaining steps according to the speculative acceptance rate.

2.2 LLM AND AGENT SCHEDULING

Existing LLM schedulers optimize individual requests through predicted output lengths or relative length rankings (Qiu et al., 2024; Fu et al., 2024), online remaining length estimation (Shahout et al., 2025), preemptive queue-based scheduling (Wu et al., 2026), or estimated future resource demands (Gong et al., 2025; Zhang et al., 2026b). These methods operate primarily at the granularity of individual LLM calls. In contrast, SPECSYS estimates the remaining steps of an entire agent task[1] across multiple LLM calls, since independently prioritizing individual calls may not align with minimizing the end-to-end JCT of agent tasks.

Existing agent schedulers exploit application dependencies and dataflows (Lin et al., 2024), completed-call history or attained service (Luo et al., 2026), historical states and predicted future stages (Ni et al., 2025), program-level execution and resource states (Kang et al., 2026), or workflow structure and cache information (Wadlom et al., 2026). However, they do not explicitly estimate the remaining work of an entire agent task. SPECSYS instead estimates the remaining number of agent steps and prioritizes tasks closer to completion, following the shortest-remaining-time-first principle at the agent-task level to better align scheduling decisions with end-to-end JCT minimization.

3 PRELIMINARY

3.1 INFERENCE SYSTEM EXECUTION MODES

In modern LLM inference systems such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024), an **inference iteration** refers to one complete scheduling-and-execution cycle. At each iteration, the **scheduler** updates the set of running requests and constructs the execution configuration for the current batch. The resulting scheduling information is then dispatched to the **worker**, which performs model execution **`Execute_Model`** followed by token processing **`Sample_Tokens`**. Accordingly, each inference iteration follows the execution structure:

$$\text{Scheduler} \rightarrow \text{Worker}\,(\texttt{Execute_Model} \rightarrow \texttt{Sample_Tokens})\,.$$

[1] In this work, we use 'agent task' and 'agent request' interchangeably.

Different generation mechanisms follow different execution paths within each inference iteration:

- **Non-SD Execution Path**. `Execute_Model` performs a standard LLM forward pass with a **decode-oriented execution pattern**, producing next-token logits for each LLM request. `Sample_Tokens` then samples one token from these logits.
- **SD Execution Path**. `Execute_Model` verifies *multiple* draft tokens in parallel using a **prefill-style execution pattern** for multi-token queries. `Sample_Tokens` then aggregates the verification results from the LLM outputs and invokes the drafter to generate the draft tokens for subsequent verification.

Consequently, non-SD and SD follow different execution paths within each inference iteration. Existing adaptive SD approaches such as vLLM's Dynamic SD and SGLang's Adaptive SD unify these two paths to support runtime adaptation. However, such unification requires additional alignment between the two execution modes or forces one mode to operate with a suboptimal computation kernel, thereby introducing unnecessary overhead (Appendix C). Throughout this paper, we use *(execution) mode* to denote the scheduler-visible runtime state indicating whether SD is enabled, and *(execution) path* to denote the corresponding worker-side implementation branch used to realize that mode. Each mode maps to a separate native execution path with its own execution flow and CUDA Graph, so switching modes in SpecSys also switches the underlying execution path.

3.2 Scheduling Objective for Agent Tasks

Unlike LLM tasks, agent tasks are typically executed as multi-step workflows consisting of multiple dependent steps. The end-to-end performance of an agent task depends on the execution of the entire workflow rather than the latency of any individual LLM call. For all serial agent workflows considered in this work, we abstract their execution as a ReAct-style pattern for ease of presentation, where LLM inference and tool execution alternate throughout the workflow (Yao et al., 2023).

We define R_i to denote each agent request indexed by i and its workflow to be an ordered sequence of dependent steps, defined as

$$R_i = \langle S_i^1, S_i^2, \ldots, S_i^{K_i} \rangle, \tag{1}$$

where K_i denotes the total number of steps of R_i. Each kth step, denoted as S_i^k, consists of an LLM execution followed by an optional tool execution before the next step can be issued.

Let $T_\pi^{\text{Queue}}(S_i^j)$ denote the waiting time of step S_i^j under scheduling policy π, $T^{\text{LLM}}(S_i^j)$ denote its LLM execution time, and $T^{\text{Tool}}(S_i^j)$ denote the tool execution time. The end-to-end Job Completion Time (JCT) of the agent request R_i under policy π can be expressed as

$$\text{JCT}_\pi(R_i) = \sum_{j=1}^{K_i} \left(T_\pi^{\text{Queue}}(S_i^j) + T^{\text{LLM}}(S_i^j) \right) + \sum_{j=1}^{K_i - 1} T^{\text{Tool}}(S_i^j). \tag{2}$$

Given a set of agent requests $\mathcal{R}$, our scheduling objective is to find a feasible scheduling policy π that minimizes the average end-to-end JCT over all agent requests in $\mathcal{R}$:

$$\pi^* = \arg\min_{\pi \in \Pi} \frac{1}{|\mathcal{R}|} \sum_{R_i \in \mathcal{R}} \text{JCT}_\pi(R_i), \tag{3}$$

where Π denotes the set of feasible scheduling policies that respect the dependency between agent steps and the serving capacity of the inference system.

This global scheduling problem is computationally challenging due to dependencies among agent steps and contention among concurrent requests. Astraea (Ni et al., 2025) formalizes a similar multi-stage agent scheduling problem and has proved it to be NP-hard. Therefore, practical online agent serving systems typically rely on efficient heuristic policies to approximate the optimal schedule.

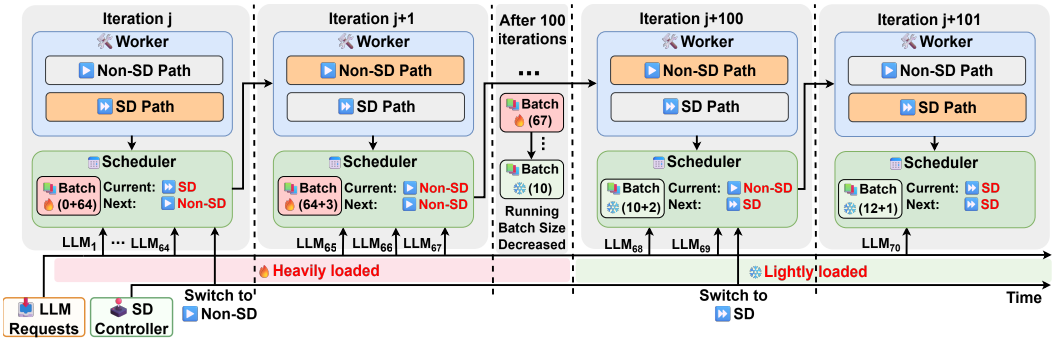

Figure 2: Illustration of iteration-level SD switching in SPECSYS. At iteration j, 64 incoming requests drive the system into a heavily loaded state, causing the external SD Controller to issue a switch-to-Non-SD request. The internal Scheduler applies the request at iteration $j+1$, and inference proceeds along the native Non-SD path for the next 100 iterations as the running batch size gradually decreases. After the system becomes lightly loaded, the SD Controller issues a switch-to-SD request, which takes effect at iteration $j + 101$, after which inference continues along the native SD path.

4 METHOD

4.1 EXTERNALLY CONTROLLABLE SD SWITCHING

Existing dynamic SD approaches have two limitations. First, their switching logic is embedded within the inference engine, limiting customized upper-layer control. Second, disabling SD does not necessarily restore the native non-SD path: as shown in Table 1, vLLM Dynamic SD (vLLM-DSD) and SGLang Adaptive SD (SGLang-ASD) with zero draft length remain token-consistent with native SD but not with native Non-SD, indicating that they still follow the SD path (detailed in Appendix C). SPECSYS addresses these issues by decoupling mode selection logic from engine execution and switching directly between native SD and non-SD paths, significantly reducing non-SD overhead (see Figure 3). As shown in Figure 2, this design consists of three components:

External Control Interface. SPECSYS exposes a lightweight control interface that receives SD switching requests and forwards them to the scheduler to update the desired execution mode. This design decouples the switching policy from internal execution logic, allowing customized policies to be changed or replaced without modifying the underlying inference engine.

Iteration-Level State Transition. Serving conditions can change while a continuous batch remains active, so coarse-grained reconfiguration may delay a needed switch. SPECSYS therefore applies mode changes at inference-iteration boundaries, so controller decisions take effect at the next iteration without disrupting the current one. The scheduler tracks current and next modes; external requests update the next mode, which is then propagated to the worker.

Path-Specific Worker Execution. Existing dynamic SD approaches focus on the SD path and provide limited optimization for native Non-SD execution, especially dedicated CUDA Graphs. Extending switching to two native paths therefore requires optimizing both paths independently. SPECSYS prepares CUDA Graphs for both paths during initialization, and the worker uses the pre-captured CUDA Graph corresponding to the selected path in each iteration.

Table 1: Token consistency across adaptive SD approaches.

Inference Engine	Approach	Non-SD	SD
vLLM	Dynamic SD (draft_len = 0)	×	✓
	SpecSys-NonSD	✓	–
	SpecSys-SD	–	✓
SGLang	Adaptive SD (draft_len = 0)	×	✓
	SpecSys-NonSD	✓	–
	SpecSys-SD	–	✓

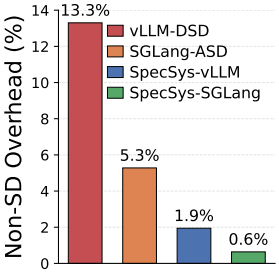

Figure 3: Non-SD Path Overhead of Adaptive SD

4.2 AGENT REMAINING-STEP PREDICTION

To exploit agent-specific execution patterns for remaining-step prediction, we profile representative agent workflows and categorize them into three types:

- **Type A** represents planning-based workflows, such as Plan-and-Solve, where the planned step count can be extracted from the planning output. A parsing example is provided in Appendix E.
- **Type B** represents multi-step iterative workflows, such as AgentFlow, where tasks terminate at specific step boundaries and exhibit a multimodal step distribution (see Appendix D), enabling completion-step estimation from historical distributions.
- **Type C** represents step-by-step workflows, such as ReAct, where tasks may terminate at almost any step, requiring additional execution-state information for prediction.

Across these workflow types, our profiling identifies two complementary signals for remaining-step prediction: historical execution patterns conditioned on workflow and workload, and execution-state cues exposed by the current LLM-call prompt.

Request-Level Information Extraction. For each LLM request, the upstream agent application provides its workflow, workload, and current execution step as metadata. SPECSYS further parses the prompt for keywords indicating *errors, repeated actions, completion, and termination*, and maps the request to a workflow-specific *execution state*. Concrete examples are provided in Appendix E.

Offline History Profiling. SPECSYS groups completed agent requests by workflow and workload and records the distribution of total execution steps T. The distributions are further conditioned on the extracted execution state and, for Type C workflows, the transition between the previous and current states. The profiling cost is negligible, as SPECSYS only maintains lightweight statistics over completed requests and performs no additional computation on the online serving path.

Remaining-Step Estimation. For each workflow type, SPECSYS estimates the remaining steps using workflow-specific signals. Let $\hat{R}_x$ denote the estimate for Type x. Type A leverages the explicit plan; Type B uses the most frequent historical completion step to capture recurring termination patterns; Type C further incorporates state transitions and uses the median for more robust estimation.

For **Type A**, let P denote the planned step count extracted from the prompt. Given the current step k, the remaining steps are directly computed as

$$\hat{R}_A = \max(0, P - k). \tag{4}$$

For **Type B**, let s_k denote the execution state at step k. SPECSYS estimates the completion step using the mode of the historical distribution conditioned on the current step and execution state:

$$\hat{T}_B = \text{mode}\{T \mid k, s_k\}, \qquad \hat{R}_B = \max(0, \hat{T}_B - k). \tag{5}$$

For **Type C**, SPECSYS further conditions the historical records on both the previous state s_{k-1} and current execution state s_k and uses the median of the corresponding historical distribution:

$$\hat{T}_C = \text{median}\{T \mid k, s_{k-1}, s_k\}, \qquad \hat{R}_C = \max(0, \hat{T}_C - k). \tag{6}$$

When the fine-grained statistics are insufficient, SPECSYS progressively falls back to the current-state-conditioned distribution and then to the workflow-workload-level historical distribution.

4.3 LOAD-ADAPTIVE AGENT SCHEDULING AND SD SELECTION

Offline SD Speedup Profiling. For each model and hardware configuration, SPECSYS profiles SD speedup across different batch sizes before engine startup and identifies the break-even load B^*, defined as the largest batch size at which SD achieves a speedup greater than or equal to 1. SD is beneficial below B^*, while its overhead can cause performance degradation above this point. Detailed profiling results are provided in Appendix B.

Future-Load Estimation. Since SD switching takes effect with a short delay, the current running batch size alone may not reflect the load when the new mode becomes active. SPECSYS therefore

also considers agent requests temporarily held by the router-proxy that are scheduled to be resumed and submitted to the inference engine. Specifically, it estimates the near-future load by combining the current running batch size B_{running} with the number of requests to be resumed B_{resume}:

$$\hat{B} = B_{\text{running}} + B_{\text{resume}}. \tag{7}$$

Load-Adaptive SD Selection. SPECSYS compares the estimated load $\hat{B}$ with the profiled break-even load B^* and sends an SD-switching request to the inference engine interface accordingly. The SD execution mode is determined as

$$\text{Enable-SD} = \begin{cases} \text{True}, & \hat{B} \leq B^*, \\ \text{False}, & \hat{B} > B^*. \end{cases} \tag{8}$$

This strategy enables SD under low load to mitigate the decode bottleneck.

Agent Task Scheduling. When the serving load becomes high and agent tasks start queuing at the router-proxy, SPECSYS prioritizes the task with the fewest predicted remaining steps:

$$j^* = \arg\min_{j \in \mathcal{Q}} \hat{R}_j, \tag{9}$$

where $\mathcal{Q}$ denotes the queued agent tasks and $\hat{R}_j$ is the remaining steps of task j estimated by the predictor in subsection 4.2. This strategy prioritizes tasks closer to completion, thereby mitigating the queue bottleneck under high serving load.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Hardware. SPECSYS is evaluated on two servers representing a resource-constrained environment and a relatively resource-abundant environment: **4× NVIDIA RTX 4090-24GB** GPUs connected via PCIe 4.0 with an Intel Xeon Gold 6338N CPU (128 cores), and **4× NVIDIA A100-SXM4-80GB** GPUs connected via NVLink 3.0 with an Intel Xeon Platinum 8369B CPU (128 cores).

Models. We evaluate SPECSYS on **Qwen3-14B**, **Qwen3-32B** (Qwen Team, 2025), and **Llama-3.3-70B-Instruct** (Grattafiori et al., 2024) to cover different model scales.

Workflows. To cover diverse agent execution patterns, we evaluate three representative workflows: **ReAct** (Yao et al., 2023), **Plan-and-Solve** (Wang et al., 2023), and **AgentFlow** (Li et al., 2026b).

Workloads. Our evaluation covers four datasets: **MBPP** (Austin et al., 2021), **2WikiMultihopQA** (Ho et al., 2020), **NL2Bash** (Lin et al., 2018), and **Spider** (Yu et al., 2018), corresponding to coding, web search, Bash, and SQL tasks, respectively.

Online Traces. Unlike evaluations based on fixed offline workloads, we use request arrival timestamps from two real-world serving traces, the **Qwen-Bailian Anonymous Dataset** (Wang et al., 2025) and **Microsoft Azure Traces** (Patel et al., 2024), to evaluate SPECSYS under dynamic serving loads. The Qwen-Bailian trace exhibits relatively smooth load variation, while the Azure trace shows more pronounced fluctuations. Details are provided in Appendix F.

Baselines. We compare SPECSYS with **vLLM Dynamic SD** and **SGLang Adaptive SD** as adaptive speculative decoding baselines, and **ThunderAgent** as an agent scheduling baseline.

Metrics. We use average **Job Completion Time (JCT)** as the primary metric for end-to-end agent latency, defined as the elapsed time from task arrival to completion of the final workflow step. Beyond average JCT, we further report P90/P95 JCT and system throughput in Appendix H, showing that SPECSYS's latency improvements do not come at the cost of tail latency or throughput.

Detailed request characteristics and hardware configurations are provided in Appendix G.

5.2 END-TO-END PERFORMANCE

We extend vLLM and SGLang with the runtime SD-switching interface (subsection 4.1) and implement SPECSYS 's completion-aware scheduling by extending ThunderAgent's scheduler (subsection 4.3). Figure 4 compares the average JCT of SPECSYS and four baselines across seven

configurations. For each configuration, the SPECSYS bar reports the lower measured average JCT of SPECSYS-vLLM and SPECSYS-SGLang.

Different Request Traces. The Azure and Qwen-Bailian traces stress different aspects of SPECSYS. Under Azure's frequent load fluctuations, SPECSYS selectively enables SD at favorable loads while disabling SD under high load, achieving **1.18–1.64**× speedup over the **best** baseline across different models and hardware configurations. Under the sustained high load of the Qwen-Bailian trace, SD provides fewer acceleration opportunities and can even increase JCT, making completion-aware scheduling more important; SPECSYS achieves **1.88**× and **2.10**× speedups on Qwen3-14B and Qwen3-32B, respectively. By completing the short-remaining tasks earlier, scheduling also reduces system load and creates opportunities for subsequent SD acceleration.

Different Models and Hardware Platforms. On the 4090 platform, we deploy Qwen3-14B and Qwen3-32B with TP=2 and TP=4, respectively, to represent resource-constrained settings, characterized by limited KV-cache capacity, lower concurrent-request capacity, lower compute capacity, and low-bandwidth inter-GPU communication. In contrast, on the A100 platform, we deploy Qwen3-14B, Qwen3-32B, and Llama-3.3-70B-Instruct with TP=1, TP=2, and TP=4, respectively, to represent resource-abundant settings, with larger KV-cache capacity, higher concurrent-request capacity, stronger compute capability, and high-speed NVLink interconnects. (See Appendix G)

- Under the resource-constrained setting, limited compute capability and memory resources introduce two challenges that SPECSYS addresses through its SD switching and completion-aware scheduling, respectively. Overall, SPECSYS achieves **1.31–2.10**× speedup under this setting.
  - The system becomes compute-bound at relatively small running batch sizes, resulting in a low break-even point B^* beyond which SD no longer provides acceleration. Consequently, non-SD (see ThunderAgent in Figure 4) is preferred for a large fraction of execution. Existing dynamic SD baselines retain additional SD-related overhead in this regime and can even perform worse than pure non-SD. In contrast, SPECSYS performs a more complete runtime switch and incurs lower overhead in non-SD mode, leading to faster inference and lower average JCT.
  - Limited KV-cache capacity constrains the runnable batch size and causes more requests to remain queued. ThunderAgent primarily optimizes spatial utilization by admitting as many requests as possible within the memory to maximize throughput, whereas SPECSYS additionally considers the temporal dimension by prioritizing jobs with shorter predicted remaining steps. This allows SPECSYS to complete jobs earlier and further reduce average JCT.
- Under the resource-abundant setting, abundant KV-cache capacity substantially reduces queueing, while its stronger compute capability increases the break-even load B^* (defined in subsection 4.3), allowing SD to remain beneficial under higher serving loads. Consequently, there is less room for both scheduling and SD switching optimizations. Nevertheless, SPECSYS still achieves the lowest average JCT, with **1.18–1.27**× speedup, demonstrating its effectiveness even when the optimization headroom is limited.

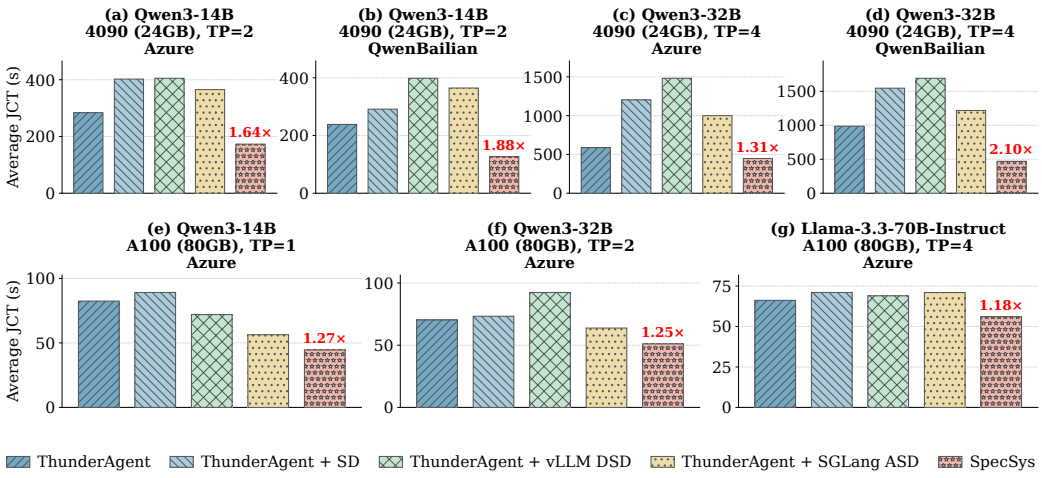

Figure 4: End-to-End average JCT across different serving configurations.

Table 2: Remaining-step prediction performance.

	Type-A	Type-B	Type-C	Overall
#Items	2167	3504	1611	7282
Pair Accuracy	1.000	0.748	0.756	0.841
Top-10% Recall	1.000	0.630	0.673	0.822

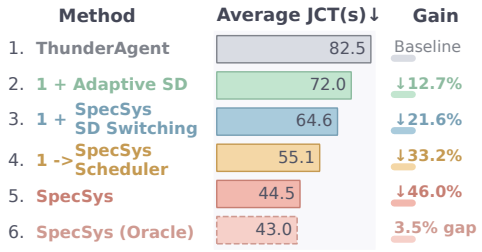

Figure 5: Ablation study of SD switching and scheduling in SPECSYS.

Beyond average JCT, we further report P90/P95 JCT and system throughput in Appendix H, showing that SPECSYS's latency improvements do not come at the cost of tail latency or throughput.

5.3 ABLATION STUDY

We conduct an ablation study on the A100 platform using Qwen3-14B with TP=1 to evaluate the effectiveness of each component in SPECSYS, following the setup of Figure 4(e).

Remaining-Step Prediction Accuracy. Since our scheduler relies on queue ranking, we evaluate the predictor in subsection 4.2 using Pair Accuracy and Top-10% Recall, which measure pairwise ordering accuracy and recall of the true shortest 10% jobs, respectively. As shown in Table 2, Type-A achieves 100% accuracy and recall since its remaining steps are explicitly available from the planning prompt, while Type-B benefits from its multimodal step distribution. Although Type-C is more difficult to predict, state information still provides useful estimates. Overall, SPECSYS achieves **84.1% Pair Accuracy** and **82.2% Top-10% Recall**, providing reliable signals for scheduling.

Impact of Adaptive SD Mechanisms. We first compare different implementations of adaptive SD. Replacing Adaptive SD with SPECSYS's SD switching reduces average JCT from 72.0 s to 64.6 s (Figure 5, 2 vs 3). This improvement mainly comes from SPECSYS's lower overhead in non-SD mode (see Appendix C), while preserving comparable performance in SD mode.

Impact of Scheduling. We next evaluate the impact of SPECSYS's completion-aware scheduling. Compared with ThunderAgent (Figure 5, 1), adding the SPECSYS scheduler (Figure 5, 4) reduces average JCT from 82.5 s to 55.1 s, corresponding to a **1.50**× speedup. Compared with using only SPECSYS's SD switching mechanism (Figure 5, 3), further incorporating SPECSYS's scheduler (Figure 5, 5) reduces average JCT from 64.6 s to 44.5 s, corresponding to a **1.45**× speedup. This improvement demonstrates that remaining-step-aware scheduling provides complementary benefits, especially under high serving load where queueing becomes a major bottleneck, by prioritizing jobs with shorter predicted remaining steps to complete them earlier and reduce queueing delay.

Gap to Oracle. Finally, we compare SPECSYS with an oracle configuration that uses oracle remaining-step information for scheduling. The oracle configuration achieves an average JCT of 43.0 s, compared with 44.5 s for SPECSYS, leaving only a **3.5%** gap (Figure 5, 5 vs 6). This small gap indicates that the predicted remaining steps are sufficient to achieve performance close to scheduling with ground-truth remaining steps.

6 CONCLUSION

In this work, we present SPECSYS, an efficient inference system that jointly optimizes speculative decoding and task scheduling for LLM-powered agents under dynamic serving loads. By exposing a low-overhead runtime interface that enables complete iteration-level switching between native SD and non-SD execution, selectively enabling SD under low serving loads, and prioritizing tasks with fewer predicted remaining steps under high serving loads, SPECSYS adapts to the changing performance bottlenecks of online agent serving to minimize average job completion time (JCT). Extensive evaluations across three LLMs, two hardware platforms, four agent workloads, three agent workflows, and two real-world online serving traces show that SPECSYS consistently improves end-to-end performance, achieving 1.18×–2.10× speedups over the best baselines.

AI USE STATEMENT

In this work, we used generative AI tools to assist with writing, information retrieval and discovery, and research ideation and execution.

For writing, generative AI was used to assist with drafting, polishing, and checking the formatting of the manuscript. For retrieval and discovery, generative AI was used to summarize literature and identify related work and relevant sources. For research ideation and execution, generative AI was used to assist with code understanding, simple script generation, code review, debugging, and identifying potential technical issues.

All AI-assisted outputs were reviewed by the authors. Suggested literature and references were checked against the original sources, AI-assisted code and debugging suggestions were manually inspected and validated through execution and testing, and AI-assisted manuscript text was reviewed and revised for technical correctness and consistency with the reported results.

We take responsibility for the final content of this work, including all text, claims, code, and other artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We make extensive efforts to facilitate the reproducibility of SpecSys. Additional details required to reproduce our experiments are provided in the appendix. In particular, Appendix K reports the agent-trace collection and replay procedure, inference-engine and generation configurations, speculative decoding settings, scheduling parameters, and software environments. Appendix E provides concrete examples and implementation details of the feature extraction and remaining-step prediction procedures for different workflow types. We further report the workload and request characteristics, hardware serving capacity, and online trace statistics in Appendix G and Appendix F. The profiling methodology and configuration-dependent thresholds used for selective SD are documented in Appendix B. Together, these descriptions specify the workloads, serving environments, system configurations, and implementation choices used throughout our evaluation. We have also released the SpecSys source code and the collected agent execution trajectories in an anonymous public repository at `https://anonymous.4open.science/r/SpecSys-Code-E03D/` to facilitate reproduction of our results.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Xin Cheng, Xingkai Yu, Chenze Shao, Jiashi Li, Yunfan Xiong, Yi Qian, Jiaqi Zhu, Shirong Ma, Xiaokang Zhang, Jiasheng Ye, et al. Dspark: Confidence-scheduled speculative decoding with semi-autoregressive generation. *arXiv preprint arXiv:2607.05147*, 2026.

Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. Efficient llm scheduling by learning to rank. *Advances in Neural Information Processing Systems*, 37:59006–59029, 2024.

Ruihao Gong, Shihao Bai, Siyu Wu, Yunqian Fan, Zaijun Wang, Xiuhong Li, Hailong Yang, and Xianglong Liu. Past-future scheduler for llm serving under sla guarantees. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 798–813, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Yilin Guan, Qingfeng Lan, Fei Sun, Dujian Ding, Devang Acharya, Chi Wang, William Wang, and Wenyue Hua. Dynamic speculative agent planning. In *International Conference on Learning Representations*, volume 2026, pp. 7695–7717, 2026.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing a multi-hop qa dataset for comprehensive evaluation of reasoning steps. In *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 6609–6625, 2020.

Yunlong Hou, Fengzhuo Zhang, Cunxiao Du, Xuan Zhang, Jiachun Pan, Tianyu Pang, Chao Du, Vincent Tan, and Zhuoran Yang. BanditSpec: Adaptive speculative decoding via bandit algorithms. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 24045–24079. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/hou25h.html.

Xinyi Hu, Yuhao Shen, Baolin Zhang, Hengxin Zhang, Jun Dai, Shuang Ge, Lei Chen, Yue Li, and Mingcheng Wan. Echo: Elastic speculative decoding with sparse gating for high-concurrency scenarios. *arXiv preprint arXiv:2604.09603*, 2026a.

Yunhai Hu, Zining Liu, Xiangyang Yin, Tianhua Xia, BO BAO, Eric Sather, Vithursan Thangarasa, and Sai Qian Zhang. DREAM-r: Multimodal speculative reasoning with RL-based refined drafting, precise verification, and fully parallel execution. In *The First Workshop on Efficient Spatial Reasoning*, 2026b. URL https://openreview.net/forum?id=CRgWv0kWjF.

Wenyue Hua, Mengting Wan, Jagannath Vadrevu, Ryan Nadel, Yongfeng Zhang, and Chi Wang. Interactive speculative planning: Enhance agent efficiency through co-design of system and user interface. In *International Conference on Learning Representations*, volume 2025, pp. 14256–14283, 2025.

Kaixuan Huang, Xudong Guo, and Mengdi Wang. Specdec++: Boosting speculative decoding via adaptive candidate lengths. In *Second Conference on Language Modeling*, 2025a. URL https://openreview.net/forum?id=Y131N9fUbU.

Kaiyu Huang, Hao Wu, Zhubo Shi, Han Zou, Minchen Yu, and Qingjiang Shi. Adaspec: Adaptive speculative decoding for fast, slo-aware large language model serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, SoCC '25, pp. 361–374. ACM, November 2025b. doi: 10.1145/3772052.3772239. URL http://dx.doi.org/10.1145/3772052.3772239.

Hao Kang, Ziyang Li, Weili Xu, Xinyu Yang, Yinfang Chen, Junxiong Wang, Beidi Chen, Tushar Krishna, Chenfeng Xu, and Simran Arora. Thunderagent: A fast, simple, and program-aware agentic inference system. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=kR4iOTaAOJ.

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*, pp. 611–626, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International conference on machine learning*, pp. 19274–19286. PMLR, 2023.

Rui Li, Zhaoning Zhang, Libo Zhang, Huaimin Wang, Xiang Fu, and Zhiquan Lai. Nightjar: Dynamic adaptive speculative decoding for large language models serving. *Journal of Systems Architecture*, pp. 103889, 2026a.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 28935–28948. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/li24bt.html.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 136737–136756. Curran Associates, Inc., 2025. doi: 10.52202/085713-4562. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/c7b5a35ea98b62512a869c19ea7b03cb-Paper-Conference.pdf.

Zhuofeng Li, Haoxiang Zhang, Seungju Han, Sheng Liu, Jianwen Xie, Yu Zhang, Yejin Choi, James Zou, and Pan Lu. In-the-flow agentic system optimization for effective planning and tool use. In *International Conference on Learning Representations (ICLR)*, 2026b.

Zikun Li, Zhuofu Chen, Remi Delacourt, Gabriele Oliaro, Zeyu Wang, Qinghan Chen, Shuhuai Lin, April Yang, Zhihao Zhang, Zhuoming Chen, et al. Adaserve: Accelerating multi-slo llm serving with slo-customized speculative decoding. In *Proceedings of the 21st European Conference on Computer Systems*, pp. 36–54, 2026c.

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)*, pp. 929–945, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/lin-chaofan.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the eleventh international conference on language resources and evaluation (LREC 2018)*, 2018.

Tianyu Liu, Yuhao Shen, Rui Cen, Junhan Shi, Jiebin Zhang, Guangshuo Qin, Hong Liu, Song Liu, Guanghua Yu, and Jianchen Zhu. D-cut: Adaptive verification depth pruning for batched speculative decoding. *arXiv preprint arXiv:2607.14647*, 2026a.

Xiaoxuan Liu, Jongseok Park, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Chen Zhang, Kuntai Du, Xiangxi Mo, Kaichao You, Alvin Cheung, et al. Turbospec: Closed-loop speculation control system for optimizing llm serving goodput. *arXiv preprint arXiv:2406.14066*, 2024.

Xiaoxuan Liu, Jiaxiang Yu, Jongseok Park, Ion Stoica, and Alvin Cheung. Speculative decoding: Performance or illusion? *Proceedings of Machine Learning and Systems*, 8:1719–1741, 2026b.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Agentix: An efficient serving engine for LLM agents as general programs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 2443–2459, Renton, WA, May 2026. USENIX Association. ISBN 978-1-939133-54-0. URL https://www.usenix.org/conference/nsdi26/presentation/luo.

Forough Majidi, Mohammad Mehdi Morovati, Foutse Khomh, and Heng Li. Llm serving in the wild: An empirical study of frameworks, methods, and system designs. *arXiv preprint arXiv:2608.03036*, 2026.

Michele Marzollo, Jiawei Zhuang, Niklas Roemer, Niklas Zwingenberger, Lorenz K Muller, and Lukas Cavigelli. Sssd: Simply-scalable speculative decoding. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33123–33139, 2026.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Hongqiu Ni, Jiabao Zhang, Guopeng Li, Zilong Wang, Ruiqi Wu, Chi Zhang, and Haisheng Tan. Astraea: A state-aware scheduling engine for llm-powered agents. *arXiv preprint arXiv:2512.14142*, 2025.

Gabriele Oliaro, Zhihao Jia, Daniel Campos, and Aurick Qiao. Suffixdecoding: Extreme speculative decoding for emerging ai applications. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 126326–126354. Curran Associates, Inc., 2025. doi: 10.52202/085713-4211. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/b7aea253ab34a773967f1e4cdea9e4fb-Paper-Conference.pdf.

Rui Pan, Yinwei Dai, Zhihao Zhang, Gabriele Oliaro, Zhihao Jia, and Ravi Netravali. Specreason: Fast and accurate inference-time compute via speculative reasoning. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 12730–12749. Curran Associates, Inc., 2025. doi: 10.52202/085713-0428. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/12c45a68e8433b21b91cd47731387fa4-Paper-Conference.pdf.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 5687–5711, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.378. URL https://aclanthology.org/2023.findings-emnlp.378.

Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. Efficient interactive llm serving with proxy model-based sequence length prediction. In *The 5th International Workshop on Cloud Intelligence / AIOps at ASPLOS 2024*, volume 5, pp. 1–7, San Diego, CA, USA, 2024. Association for Computing Machinery.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Rana Shahout, Eran Malach, Chunwei Liu, Weifan Jiang, Minlan Yu, and Michael Mitzenmacher. Don't stop me now: Embedding based scheduling for llms. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 63345–63368, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/9eb8b5ccb0de594a16548f7c058fdadf-Paper-Conference.pdf.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Noppanat Wadlom, Junyi Shen, and Yao Lu. Efficient llm serving for agentic workflows: A data systems perspective. *Proc. ACM Manag. Data*, 4(3), May 2026. doi: 10.1145/3802046. URL https://doi.org/10.1145/3802046.

Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. Kvcache cache in the wild: Characterizing and optimizing kvcache cache at a large cloud provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, July 2025. URL https://www.usenix.org/conference/atc25/presentation/wang-jiahao.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 2609–2634, 2023.

Xin Wang, Ziming Miao, Yi Zhu, Hui Shen, Zhongwei Wan, Fan Yang, and Mi Zhang. Agentspec: Speculative decoding for batch inference of llm agents. In *EMNLP 2026*. EMNLP, August 2026. URL https://www.microsoft.com/en-us/research/publication/agentspec-speculative-decoding-for-batch-inference-of-llm-agents/.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. FastServe: Iteration-Level preemptive scheduling for large language model inference. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 57–74, Renton, WA, May 2026. USENIX Association. ISBN 978-1-939133-54-0. URL https://www.usenix.org/conference/nsdi26/presentation/wu-bingyang.

Heming Xia, Yongqi Li, Cunxiao Du, Mingbo Song, and Wenjie Li. Toolspec: Accelerating tool calling via schema-aware and retrieval-augmented speculative decoding. *arXiv preprint arXiv:2604.13519*, 2026.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854, 2023.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

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 *International Conference on Learning Representations*, 2023. URL https://mlanthology.org/iclr/2023/yao2023iclr-react/.

Naimeng Ye, Arnav Ahuja, Georgios Liargkovas, Yunan Lu, Kostis Kaffes, and Tianyi Peng. Speculative actions: A lossless framework for faster AI agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=P0GOk5wslg.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 3911–3921, 2018.

Jiebin Zhang, Zhenghan Yu, Liang Wang, Nan Yang, Eugene Yu, Zheng Li, Yifan Song, Dawei Zhu, Xingxing Zhang, Furu Wei, et al. Learning to draft: Adaptive speculative decoding with reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 30798–30819, 2026a.

Wei Zhang, Zhiyu Wu, Yi Mu, Rui Ning, Banruo Liu, Nikhil Sarda, Myungjin Lee, and Fan Lai. JITServe: SLO-aware LLM serving with imprecise request information. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 825–848, Renton, WA, May 2026b. USENIX Association. ISBN 978-1-939133-54-0. URL https://www.usenix.org/conference/nsdi26/presentation/zhang-wei.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

A LLM TIME BREAKDOWN AND BOTTLENECK ANALYSIS IN AGENT TASKS

Table 3: LLM inference time breakdown across agent tasks.

Task	Dataset	Avg. JCT (s)	Avg. LLM Time (s)	LLM Time Ratio
Web Search	2WikiMultihopQA	60.36	36.49	60.46%
Web Search	Bamboogle (Press et al., 2023)	49.15	32.91	66.95%
Web Search	HotpotQA (Yang et al., 2018)	40.72	29.14	71.56%
Coding	MBPP	30.10	30.06	99.87%

We measure the end-to-end execution time and the fraction spent on LLM inference across different agent tasks and datasets, as shown in Table 3. LLM inference accounts for the majority of the end-to-end latency across all evaluated settings. For some tasks, the execution time of external tools is negligible, making LLM inference responsible for nearly all of the overall latency. These results indicate that the end-to-end latency of agent tasks is primarily bottlenecked by LLM inference. Coding and Web Search tasks typically represent two contrasting cases. In Coding tasks, the tool execution mainly consists of running generated code and is usually very fast, whereas in Web Search tasks, the agent often needs to retrieve and access multiple webpages, making tool execution more sensitive to network latency. Despite this difference, LLM inference still accounts for a substantial fraction of the end-to-end latency in both types of tasks.

We further analyze the latency breakdown of LLM requests in agent tasks, as shown in Figure 6. The latency of an LLM request mainly consists of three components: prefill, decode, and queueing time. Existing serving systems have extensively optimized KV-cache reuse, substantially reducing redundant computation during the prefill stage and making prefill latency relatively small. To preserve KV-cache residency and avoid excessive eviction or recomputation, these systems typically employ admission control based on the number of concurrent requests and the available KV-cache capacity. When the number of concurrent requests exceeds the capacity that can be supported by the available KV cache, newly arrived requests are queued and can enter inference only after running requests complete and release their KV-cache space.

Based on these results, we observe a concurrency-dependent shift in the LLM inference bottleneck of agent tasks. At low concurrency, decode time dominates the request latency and serves as the primary performance bottleneck. As concurrency increases, more requests are blocked by admission control, causing queueing time to grow rapidly and eventually dominate the overall latency. As a result, the performance bottleneck shifts from decoding at low serving loads to queueing at high serving loads.

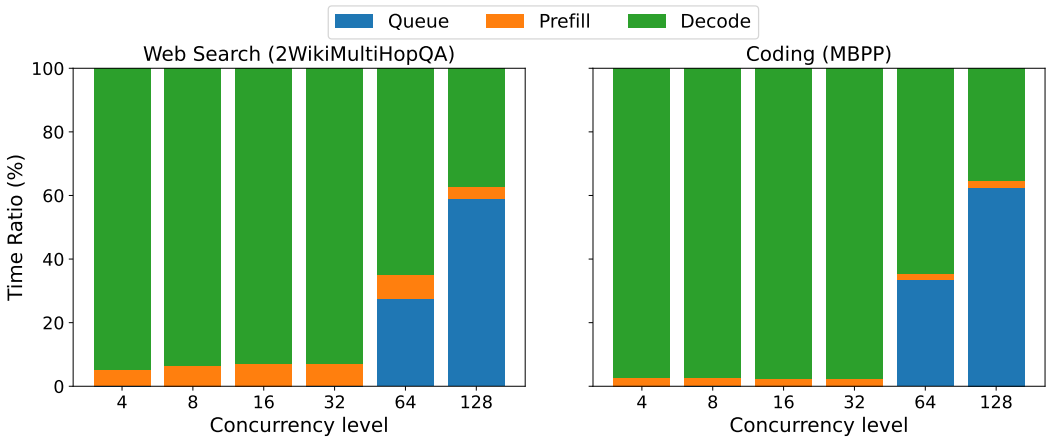

Figure 6: LLM inference time breakdown under different concurrency levels for two agent tasks, Web Search on 2WikiMultihopQA and Coding on MBPP, showing a shift from decode-dominated latency at low concurrency to queue-dominated latency at high concurrency.

B Impact of Batch Size on SD Speedup for Agent Tasks

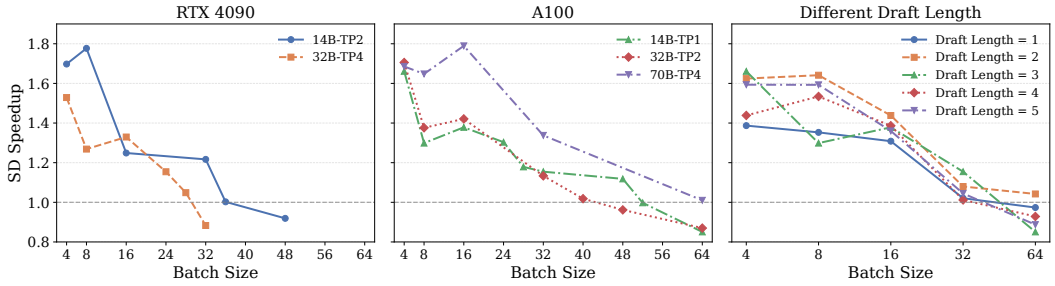

Figure 7: SD speedup under different serving loads, model configurations, and draft lengths. Across both RTX 4090 and A100 GPUs, the benefit of SD generally decreases as batch size increases and can eventually become negative, while the optimal draft length also varies with serving load.

To understand how serving load affects the effectiveness of speculative decoding, we measure the SD speedup across different batch sizes, GPU platforms, model configurations, and draft lengths, as shown in Figure 7. Across all evaluated configurations, the benefit of SD consistently decreases as the batch size increases and eventually becomes negative under sufficiently high serving loads.

Our experiments further reveal substantially different SD break-even points across hardware and model configurations. Specifically, the break-even batch sizes are 36 for 4090-14B-TP2 and 28 for 4090-32B-TP4, compared with 52 for A100-14B-TP1, 40 for A100-32B-TP2, and 64 for A100-70B-TP4. These results show that the serving load at which SD ceases to provide a performance benefit varies considerably across deployment configurations, highlighting the need to determine when SD should be enabled based on the specific serving environment.

We further observe that the speedup decreases more rapidly on RTX 4090 GPUs than on A100 GPUs. In particular, A100 GPUs maintain positive SD speedup at larger batch sizes, resulting in a higher break-even batch size between SD and non-SD execution. This observation suggests that the effective operating range of SD depends on the underlying hardware capability: more powerful GPUs can sustain the benefit of SD under higher serving loads, whereas less powerful GPUs enter the negative-speedup regime earlier. Consequently, adaptive SD selection has greater optimization opportunities on resource-constrained GPUs, where choosing between SD and non-SD execution is more critical as the serving load changes.

Different draft lengths exhibit the same overall trend. At low batch sizes, more aggressive configurations with longer draft lengths can achieve higher speedup, while increasing the batch size favors more conservative configurations with shorter draft lengths. This load-dependent behavior motivates existing adaptive SD approaches to adjust the draft length according to the current serving load. However, draft-length adaptation alone is insufficient at high loads: once the batch size becomes sufficiently large, even the most conservative configuration with a draft length of 1 can incur negative speedup. Therefore, an effective adaptive SD mechanism should not only adapt the draft length, but also be able to completely disable SD and switch to the native non-SD execution path when speculative decoding is no longer beneficial.

C Comparison of Adaptive SD Approaches

Table 4: Token consistency comparison across different approaches of adaptive SD.

Inference Engine	Engine A	Engine B	Token Consistency
vLLM	Dynamic SD (draft_len $= 0$)	non-SD	$\times$
	Dynamic SD (draft_len $= 0$)	SD	✓
	SpecSys non-SD Mode	non-SD	✓
	SpecSys-SD Mode	SD	✓
SGLang	Adaptive SD (draft_len $= 0$)	non-SD	$\times$
	Adaptive SD (draft_len $= 0$)	SD	✓
	SpecSys non-SD Mode	non-SD	✓
	SpecSys-SD Mode	SD	✓

Token Consistency across Adaptive SD Approaches. Online inference engines typically employ different execution paths for SD and non-SD generation to improve efficiency, with different kernels used for key computations such as attention. As a result, even under deterministic decoding settings (i.e., temperature $= 0$ and top-k $= 1$), SD and non-SD execution may generate slightly different token sequences for the same prompt due to differences in their underlying computation paths, although the generated contents are usually semantically similar. Therefore, token-level consistency provides a practical indicator of whether two execution modes follow the same underlying inference path. We evaluate this behavior for two adaptive SD approaches, vLLM Dynamic SD and SGLang Adaptive SD, as shown in Table 4. For both vLLM Dynamic SD and SGLang Adaptive SD, when SD is disabled by setting draft_len $= 0$, the generated tokens are inconsistent with those of the native non-SD engine but remain consistent with those produced by the native SD engine. This indicates that setting draft_len $= 0$ does not fully switch execution back to the native non-SD path; instead, the engine continues to follow the SD execution path with SD disabled. In contrast, SpecSys's non-SD mode produces tokens consistent with the native non-SD engine, while its SD mode remains consistent with the native SD engine. These results demonstrate that SpecSys performs a more complete runtime switch between the native SD and non-SD execution paths.

non-SD Overhead across Adaptive SD Approaches. Since vLLM Dynamic SD and SGLang Adaptive SD continue to execute along the SD path when draft_len $= 0$, this path is suboptimal for autoregressive non-SD generation and introduces additional performance overhead. We therefore measure the non-SD execution overhead of these two Adaptive SD approaches and SpecSys relative to the native non-SD engine, as shown in Figure 8. However, SpecSys consistently incurs substantially lower overhead than vLLM Dynamic SD and SGLang Adaptive SD across all batch sizes. These results further demonstrate the benefit of SpecSys's more complete runtime switching between the native SD and non-SD execution paths.

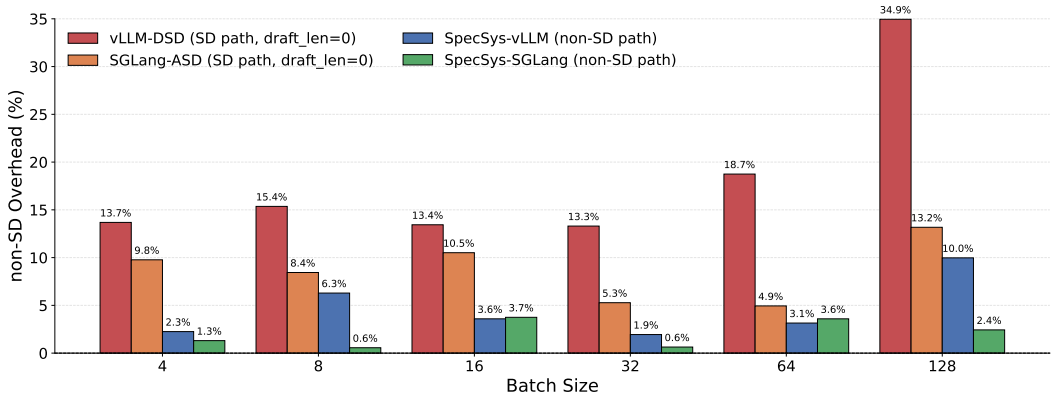

Figure 8: non-SD execution overhead across different adaptive SD approaches and batch sizes.

D AGENT STEP DISTRIBUTIONS ACROSS WORKFLOWS AND WORKLOADS

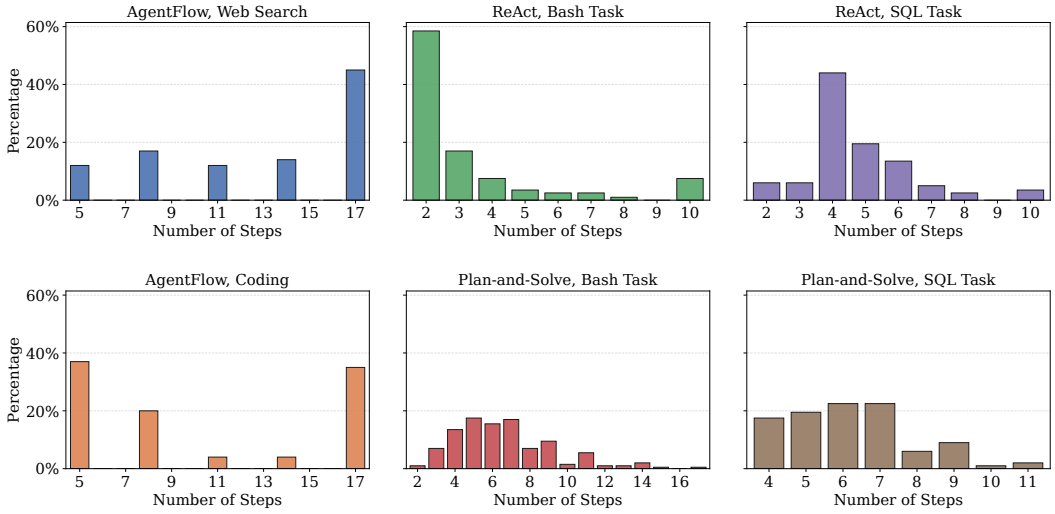

Figure 9: Distribution of agent execution steps across different workflow–workload combinations.

To investigate how agent step distributions relate to workloads and workflows, we evaluate six workload–workflow combinations, as shown in Figure 9. For the same workload, different workflows exhibit substantially different step distributions; similarly, under the same workflow, changing the workload also leads to noticeable distribution shifts. These observations suggest that agent step distributions are closely associated with the workload–workflow combination.

To further validate this relationship, we compare step distributions across different datasets while keeping both the workload and workflow fixed, as shown in Figure 10. Despite using different datasets, the resulting step distributions remain broadly similar, further supporting our observation that the workload–workflow combination is a primary factor shaping the step distribution.

Moreover, we observe a distinctive multimodal pattern for AgentFlow, a multi-step iterative workflow, where agent tasks tend to terminate at only a few specific step counts.

This observation provides an opportunity to predict the remaining steps of an agent task and motivates our prediction method to model step distributions separately for different workflow–workload combinations.

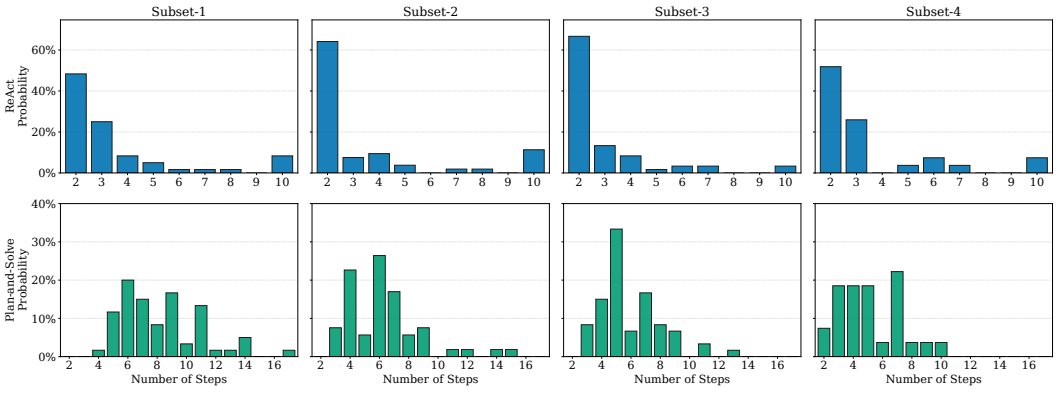

Figure 10: Distribution of agent execution steps across datasets under the same workflow and task.

E EXAMPLES OF PROMPT PARSING

E.1 EXAMPLE OF EXTRACTION FOR TYPE-A WORKFLOWS

The following prompt is taken from an execution trajectory generated by the Plan-and-Solve agent workflow on the SQL task:

```
{
  "role": "user",
  "content": "Question: Find the first names that are used for
     professionals or owners but are not used as dog names."
}
{
  "role": "assistant",
  "content": "### Understanding the Problem:\n"
             "We are asked to find first names that are used for "
             "professionals or owners but are not used as dog names.\n\n"
             "---\n\n"
             "### Plan:\n"
             "1. Check what tables are available to use.\n"
             "2. Identify the tables that contain information about "
             "professionals, owners, and dog names.\n"
             "3. Write a query to extract all unique first names from "
             "the professionals/owners table.\n"
             "4. Write a query to extract all unique dog names from "
             "the dog names table.\n"
             "5. Write a query to find the first names that are in the "
             "professionals/owners table but not in the dog names table.
             "### Step-by-Step Execution"
}
```

For this type of agent workflow, the first LLM call generates an explicit plan, which is then included in the context of subsequent LLM calls during task execution. Therefore, the total number of planned steps can be directly extracted from the planning content. Specifically, we locate the `Plan:` marker and count the numbered items in the plan. In this example, five numbered items are identified, and thus `num_plan_steps` is set to 5.

Since the upper-layer proxy can determine which LLM requests belong to the same Agent Task, the extracted `num_plan_steps` is bound to the corresponding task after it is first obtained. For subsequent LLM requests of the same task, the proxy directly retrieves this value and combines it with the `cur_step` carried by the task to estimate the number of remaining steps, without repeatedly parsing the planning content.

E.2 EXAMPLE OF EXTRACTION FOR TYPE-B WORKFLOWS

Unlike Type-A, Type-B workflows do not produce a fixed plan whose length can be directly extracted. Instead, the proxy determines a coarse-grained execution state from the latest action and observation contained in each prompt.

Example 1: Initial State. The following example is taken from AgentFlow execution trajectories on the Web Searching tasks. The first request contains the original query but no previously executed action or observation:

```
{
"query": "Which film has the director who is older than the other, Let's
    Make Money or Short Term 12?",
"available_tools":
    ["Generalist_Solution_Generator_Tool","Local_Web_Search_Tool"]
}
```

Since no action has been completed, the proxy assigns the special state `start` to this request.

Example 2: Problem State. The following example is taken from AgentFlow execution trajectories on the Web Searching tasks. The next prompt contains the result of an unsuccessful search:

```
{
"query": "Which film has the director who is older than the other, Let's
    Make Money or Short Term 12?",
"context": "Query: Let's Make Money director age, Short Term 12 director
    age",
"sub_goal": "Find the directors of Let's Make Money and Short Term 12
    along with their ages.",
"tool_name": "Local_Web_Search_Tool",
"command": "execution = tool.execute(
query="Let's Make Money director age",
result_num=3
)
execution = tool.execute(
query="Short Term 12 director age",
result_num=3
)",
"result": [
"No results found."
]
}
```

The phrase `No results found` matches the problem-related keyword `no results`, indicating that the latest action fails to obtain the expected information. Consequently, the proxy classifies the current state as `problem`. More generally, the problem state captures action outcomes that indicate unsuccessful execution, missing information, or invalid intermediate results. Other expressions that trigger this state include `error`, `failed`, `not found`, `unable`, `cannot`, `invalid`, `exception`, and `traceback`.

Example 3: Positive State. The following example is taken from AgentFlow execution trajectories on the Coding tasks. The latest tool execution indicates that the generated implementation passes all provided tests:

```
{
"query": "Write a Python function to insert a given string at the
    beginning of all items in a list.",
"sub_goal": "Generate a Python function called add_string that uses {0}
    as a placeholder for each list item.",
"tool_name": "Python_Code_Generator_Tool",
"command": "execution = tool.execute(
query="Write the add_string function and verify it using the provided
    tests."
)",
"result": {
"printed_output": "All test cases passed successfully.",
"execution_code": "def add_string(input_list, format_str):\n"
"    return [format_str.format(item) "
"for item in input_list]"
}
}
```

The result contains both `passed` and `successfully`, which match the positive-state keywords and indicate that the latest action has successfully achieved its intended sub-goal. The proxy therefore assigns the current state as `positive`. More generally, the positive state captures action outcomes that indicate successful execution, sufficient information, or progress toward task completion. Other expressions associated with this state include `verified`, `sufficient`, `answer`, and `final`.

Example 4: Neutral State. The following example is taken from AgentFlow execution trajectories on the Web Searching tasks. In the following prompt, the latest search action does not contain any predefined problem-related or positive-completion keywords:

```
{
"query": "Which film has the director who is older than the other, Let's
    Make Money or Short Term 12?",
"context": "Query: Short Term 12 director age",
"sub_goal": "Find the director and age of Short Term 12 using a more
    specific search query.",
"tool_name": "Local_Web_Search_Tool",
"command": "execution = tool.execute(
query="Short Term 12 director age",
result_num=5
)",
"result": [
"Title: Destin Daniel Cretton - Wikipedia\n"
"Snippet: Destin Yori Daniel Cretton "
"(born November 23, 1978) is an American filmmaker.\n\n"
"Title: Short Term 12 - Wikipedia\n"
"Snippet: Short Term 12 is a 2013 American independent "
"drama film written and directed by Destin Daniel Cretton."
]
}
```

Unlike the previous two states, the `neutral` state is not triggered by a dedicated set of keywords. Instead, it serves as the default state when the extracted content contains neither problem-related nor positive-completion keywords.

For each request, the proxy collects the latest action-related content, including the tool name, subgoal, command, and observation, and concatenates these fields into a single text representation for state extraction. It then searches this text for the predefined problem-related and positive-completion keywords. Problem keywords are checked first to ensure that failed or abnormal tool executions are identified even when the same content also contains seemingly positive expressions. If no problem keyword is found, the proxy checks for positive keywords that indicate successful execution, sufficient information, or progress toward task completion. When neither type of keyword is detected, the current request is classified as `neutral`. In this way, each LLM request is mapped to one of the three task-independent states according to the outcome of its latest action.

Since the upper-layer proxy can identify which LLM requests belong to the same Agent Task, it maintains a separate state sequence for each task. After the state of a newly arrived request is extracted, the state is appended to the sequence associated with that task. As the task proceeds through multiple LLM requests and tool executions, the sequence therefore incrementally records its execution progress. For example, a task may produce the following state sequence:

```
["start", "neutral", "problem", "neutral", "positive"]
```

Here, `start` denotes the initial request, while the subsequent states summarize the outcomes of successive actions. Such a sequence provides a compact representation of the task's execution history without relying on task-specific semantic information.

To estimate the number of remaining steps, the proxy compares the current task state with historical Type-B trajectories from the same workflow and workload. Specifically, it retrieves historical trajectories that reached the same extracted state at the same execution position as the current task. These matched trajectories provide empirical information about how many total steps tasks with similar execution progress typically require. If at least three matching trajectories are available, the proxy examines their endpoints and uses the most frequent endpoint as the estimated total number of steps. Using multiple matched trajectories avoids relying on a single potentially atypical execution.

When fewer than three matching trajectories are available, the proxy falls back to the unconditional step distribution of the same workflow and workload. The remaining steps are then estimated by subtracting the current step from the predicted total number of steps.

E.3 Example of Extraction For Type-C Workflows

Unlike Type-B, Type-C workflows follow a fine-grained ReAct pattern, where LLM requests interleave Thought, Action, and Observation steps. The proxy considers execution errors, repeated actions, and explicit task termination, producing five task-independent states: `start`, `normal`, `error`, `repeat`, and `final_or_submit`.

Example 1: Initial State. The following example is taken from ReAct on SQL Task. The first request contains the original question but no completed action or observation:

```
{
"role": "user",
"content": "Question: What are different nationalities of people and the
    corresponding number of people from each nation?"
}
```

Since no action has been completed, the proxy assigns the state `start`.

Example 2: Normal State. The following example is taken from ReAct on SQL Task. In the following prompt, the latest action executes successfully and returns a valid observation:

```
{
"role": "user",
"content":
"Question: What are different nationalities of people and "
"the corresponding number of people from each nation?\n"
"Thought 4: The available information allows me to perform "
"the next operation.\n"
"Action 4: execute[SELECT Nationality, COUNT(*) "
"FROM people GROUP BY Nationality]\n"
"Observation 4: [('Russia', 6), ('Bulgaria', 1)]"
}
```

Because the action succeeds, contains no execution error, and does not repeat the preceding action, the state is classified as `normal`.

Example 3: Error State. The following example is taken from ReAct on SQL Task. In the next prompt, the agent attempts to access a nonexistent table:

```
{
"role": "user",
"content":
"Question: What are different nationalities of people and "
"the corresponding number of people from each nation?\n"
"Thought 1: I should inspect the countrylanguage table "
"to determine whether it contains nationality information.\n"
"Action 1: execute[DESC countrylanguage]\n"
"Observation 1: Error executing query: "
"Table 'poker_player.countrylanguage' "
"doesn't exist"
}
```

The expressions `Error executing` and `doesn't exist` indicate that the latest action failed during execution and therefore trigger the `error` state. The proxy detects such failures from the returned observation using a predefined set of error-related keywords and patterns. Other representative indicators include `does not exist`, `no such`, `unknown column`, `syntax error`, `traceback`, and `exception`. This allows execution failures from different tools and task domains to be mapped to the same task-independent state.

Example 4: Repeated-Action State. The following example is taken from ReAct on SQL Task. In the following example, the same action is executed in two consecutive steps:

```
{
"role": "user",
"content":
"Question: Which owner has paid the largest amount of money "
"in total for their dogs?\n"
"Thought 6: I should inspect the available information.\n"
"Action 6: execute["DESC treatments"]\n"
"Observation 6: [('treatment_id', 'int'), "
"('dog_id', 'int'), ('cost_of_treatment', 'decimal')]\n"
"Thought 7: I need to inspect the relationship again.\n"
"Action 7: execute["DESC treatments"]\n"
"Observation 7: [('treatment_id', 'int'), "
"('dog_id', 'int'), ('cost_of_treatment', 'decimal')]"
}
```

After normalization, `Action 7` is identical to `Action 6`, so the proxy assigns the state `repeat`.

Example 5: Final-or-Submit State. The following example is taken from ReAct on SQL Task. Once the agent determines that the task has been completed, it submits the result:

```
{
"role": "user",
"content":
"Question: What are different nationalities of people and "
"the corresponding number of people from each nation?\n"
"Thought 4: I can count the number of people per nationality.\n"
"Action 4: execute[SELECT Nationality, COUNT(*) "
"FROM people GROUP BY Nationality]\n"
"Observation 4: [('Russia', 6), ('Bulgaria', 1)]\n"
"Thought 5: The result completely answers the question.\n"
"Action 5: submit"
}
```

Because the latest action contains `submit`, the proxy assigns the state `final_or_submit`. Actions containing `final`, or results explicitly marked as completed, are treated similarly.

For each request, the proxy extracts the latest action, its observation, and the preceding action, and assigns the state according to the following priority:

```
final_or_submit > error > repeat > normal
```

The extracted states are incrementally appended to the corresponding Agent Task, forming sequences such as:

```
["start", "error", "normal",
"repeat", "normal", "final_or_submit"]
```

Unlike Type-B, Type-C additionally conditions prediction on the transition between the previous and current states. The proxy first retrieves historical trajectories with the same previous state, current state, and execution position. If at least three matches are available, their endpoint distribution is used for prediction. Otherwise, it falls back to trajectories with the same current state and execution position, and then to the unconditional step distribution of the same workflow and workload if necessary. The median endpoint is used as the estimated total number of steps, from which the current step is subtracted to obtain the remaining-step estimate.

F REQUEST ARRIVAL PATTERNS OF ONLINE SERVING TRACES

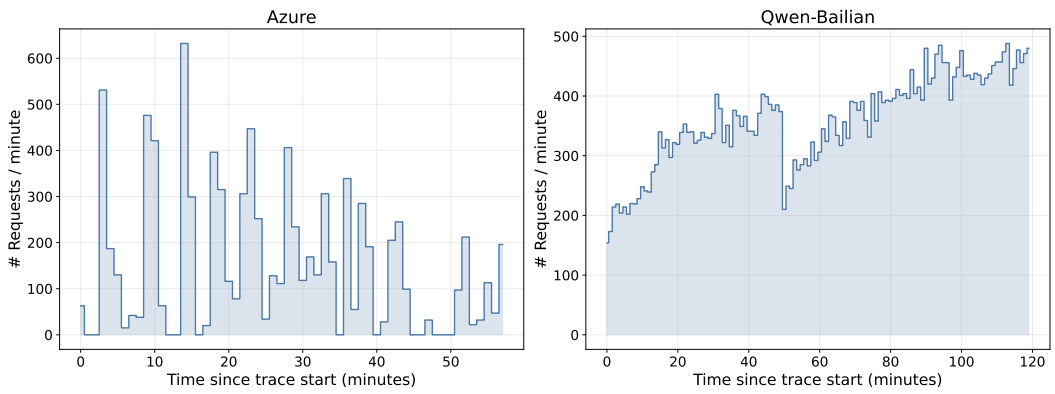

Figure 11: Request arrival patterns of the Azure and Qwen-Bailian traces, measured as the number of requests per minute over time.

We further examine the request-volume dynamics of the two online inference traces used in our end-to-end evaluation, as shown in Figure 11, and summarize their key statistics in Table 5.

Azure Trace. The Azure trace exhibits highly bursty traffic, with frequent alternations between low-load periods and sharp traffic spikes. Its average arrival rate is 152.05 requests/min, while the peak reaches 632 requests/min, about 4.16× the average. Such rapid load fluctuations make SD beneficial during low-load periods, but may cause performance degradation when sudden bursts push the serving system into a high-load regime. Therefore, this trace is well suited for evaluating whether adaptive SD methods can effectively adapt their SD decisions to rapidly changing serving loads.

Qwen-Bailian Trace. In contrast, the Qwen-Bailian trace is considerably smoother and remains under sustained high load for most of its duration. Its average arrival rate reaches 358.82 requests/min, more than twice the 152.05 requests/min observed in the Azure trace, while its peak is 488 requests/min. Under such consistently high load, the benefit of SD is limited and can even cause performance degradation, while a large number of requests may accumulate in the waiting queue. This trace is therefore well suited for evaluating both the non-SD overhead introduced by adaptive SD mechanisms under high load and the effectiveness of task scheduling strategies in reducing queueing delay.

Table 5: Statistics of the Azure and Qwen-Bailian traces.

Trace	# Requests	Duration (min)	Avg. Req./min	Peak Min.	Peak Req.
Azure	8,819	57.27	152.05	14	632
Qwen-Bailian	43,058	119.99	358.82	113	488

G AGENT REQUEST CHARACTERISTICS AND SYSTEM CAPACITY

Table 6: Statistics of different agent workload–workflow combinations.

Workflow	Workload	Traces	Avg. Prompt Tokens/Request	Avg. Steps
AgentFlow	Web Search	100	1,502	12.84
AgentFlow	Coding	100	2,133	10.21
Plan-and-Solve	Bash	200	786	6.63
Plan-and-Solve	SQL	200	627	6.21
ReAct	Bash	200	2,293	3.32
ReAct	SQL	200	2,422	4.79
Overall	All	1,000	1,504	6.49

Agent Request Analysis. We summarize the characteristics of the agent execution traces used in our experiments in Table 6. In total, the traces contain 1,000 execution traces across six workload–workflow combinations, with an average of 1,504 prompt tokens per request and 6.492 execution steps per task. The request characteristics vary substantially across different combinations. AgentFlow exhibits the largest average number of execution steps, reaching 12.84 steps on Web Search and 10.21 steps on Coding. In contrast, ReAct requires only 3.315 and 4.785 steps on Bash and SQL respectively, while having relatively long prompts of more than 2,000 tokens per request. Plan-and-Solve lies between these two workflows in terms of execution steps, while producing the shortest prompts among the evaluated combinations. These statistics demonstrate the diversity of request lengths and execution patterns covered by our experimental workloads.

Table 7: Memory usage and serving capacity under different configurations.

Configuration	Weights/GPU	KV Cache/GPU	KV Tokens	Requests Supported
4090-14B-TP2	15.25 GiB	4.45 GiB	56,944	38
4090-32B-TP4	16.33 GiB	3.38 GiB	54,544	36
A100-14B-TP1	30.12 GiB	36.72 GiB	234,768	156
A100-32B-TP2	32.08 GiB	33.96 GiB	275,632	183
A100-70B-TP4	33.93 GiB	32.10 GiB	415,525	276

Hardware Resource Configuration Analysis. We further profile the inference-engine resource characteristics under the five experimental configurations, as shown in Table 7. The number of supported requests is estimated by dividing the KV token capacity by the average prompt length reported in Table 6. On the RTX 4090 platform, the two configurations support only 36–38 concurrent requests, mainly due to the limited KV-cache capacity under the relatively tight GPU memory budget. These configurations therefore represent a resource-constrained serving environment. In contrast, the A100 configurations support approximately 156–276 requests, benefiting from their substantially larger GPU memory and KV-cache capacity, and thus represent a resource-abundant serving environment. Overall, these measurements are consistent with our experimental design, where the RTX 4090 serves as a resource-constrained configuration, while the A100 serves as a resource-abundant configuration.

H ADDITIONAL END-TO-END EVALUATION METRICS

Beyond average JCT, we further evaluate SPECSYS using additional end-to-end metrics, including P90 and P95 JCT as well as system throughput. These metrics provide a more comprehensive view of whether the average-JCT improvement of SPECSYS comes at the cost of tail latency or overall serving efficiency. The results are shown in Figure 12.

Tail Latency. Since SPECSYS reduces average JCT by approximating shortest-remaining-time-first scheduling, long-running jobs may receive lower scheduling priority, potentially increasing tail latency. To examine whether this introduces a long-tail effect, we compare the P90 and P95 JCT across different model and tensor-parallel configurations. As shown in Figure 12, SPECSYS achieves P90 and P95 JCT comparable to or lower than the baselines, and in several configurations provides noticeable reductions. This indicates that SPECSYS reduces average JCT without introducing additional tail-latency degradation.

Throughput. ThunderAgent's scheduling policy is primarily designed to maximize system throughput, whereas SPECSYS focuses on reducing average JCT through remaining-step-aware scheduling. We therefore further evaluate whether this latency improvement comes at the cost of throughput. As shown in Figure 12, SPECSYS maintains throughput comparable to the baselines across all evaluated configurations. These results demonstrate that the reduction in average JCT achieved by SPECSYS does not come at the expense of system throughput.

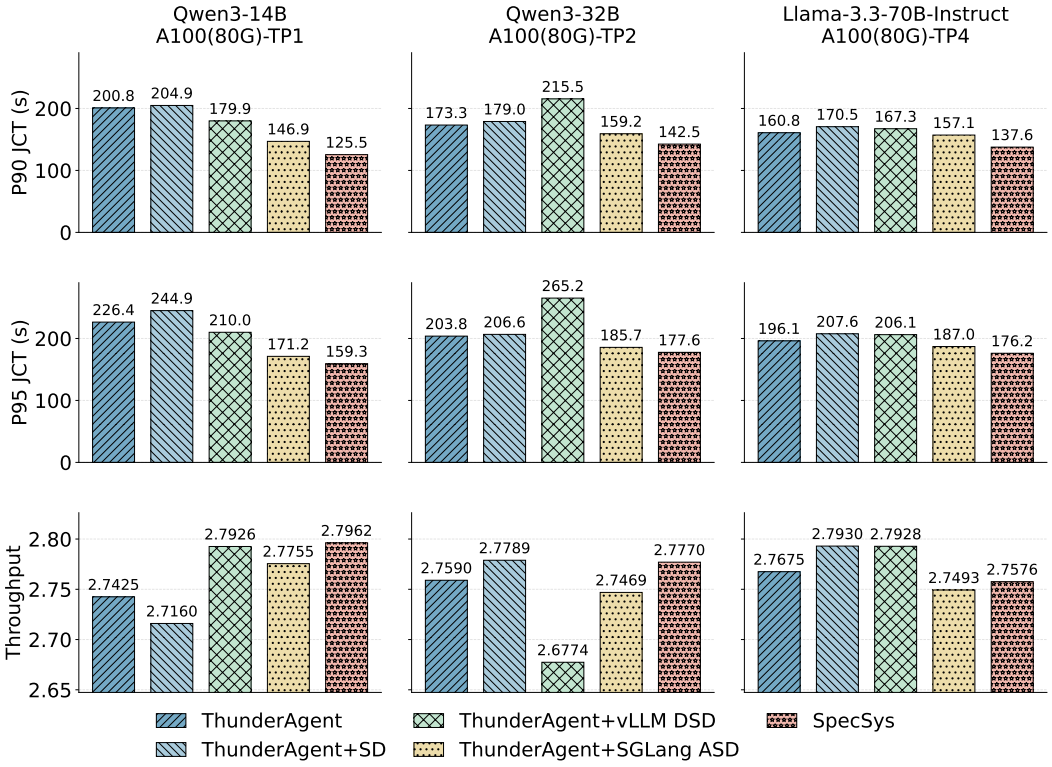

Figure 12: Additional end-to-end evaluation metrics on A100 GPUs. Across different models and tensor-parallel configurations, SPECSYS achieves lower or comparable P90 and P95 JCT while maintaining comparable throughput, showing that its average-JCT improvements do not come at the cost of tail latency or system throughput.

I OVERHEAD OF SPECSYS PREDICTION AND SCHEDULING

Table 8: Per-request scheduling overhead of ThunderAgent and SPECSYS

Scheduling Method	Processing Time (ms/request)	Increase over ThunderAgent (ms)
ThunderAgent	0.05086	–
SPECSYS	0.13189	0.08103

Table 8 compares the per-request scheduling overhead of ThunderAgent and SPECSYS. ThunderAgent requires an average of 0.05086 ms per request, while SPECSYS requires 0.13189 ms because it additionally estimates the remaining execution steps. The absolute increase is only 0.08103 ms per request. Compared with agent inference and tool-execution latencies, which typically range from hundreds of milliseconds to several seconds, this additional overhead is negligible.

Table 9: Paused-queue sorting overhead of ThunderAgent and SPECSYS

Queue Size	ThunderAgent (ms)	SPECSYS (ms)	Increase (ms)
10	0.0059	0.0084	0.0024
100	0.0482	0.0706	0.0224
1,000	0.4815	0.7855	0.3040
10,000	5.8395	9.0944	3.2549

Table 9 reports the sorting overhead under different paused-queue sizes. SPECSYS is slightly more expensive than ThunderAgent because it prioritizes programs according to their predicted remaining steps. With 1,000 paused programs, SPECSYS adds only 0.304 ms over ThunderAgent. Even with 10,000 paused programs, the additional sorting time is approximately 3.25 ms. Since ThunderAgent performs periodic scheduling at a configurable interval, this sorting overhead remains small relative to typical scheduling intervals.

J DISCUSSION OF VLLM AND SGLANG

In our experiments (Figure 4, Figure 8 and Figure 12), SGLang achieves lower inference latency than vLLM in most evaluated configurations, consistent with its aggressive runtime optimizations for high-performance LLM serving. Nevertheless, SPECSYS is also implemented on top of vLLM, and our evaluation retains vLLM-based baselines, because vLLM provides more consistent support across heterogeneous model and SD configurations, making it necessary for evaluating configurations that cannot be directly deployed on SGLang.

A recent empirical study of LLM serving frameworks (Majidi et al., 2026) shows that vLLM is the most popular and widely adopted serving framework among the studied systems, with substantially more open-source repositories adopting it than SGLang and other alternatives. The study also finds that vLLM is used across a broader range of model families, suggesting broader compatibility across heterogeneous LLMs. These observations are consistent with our deployment experience, where vLLM provides more consistent out-of-the-box support across different models and SD configurations.

In our practical deployments, using Qwen3-32B and Llama-3.3-70B-Instruct with SGLang encountered several model- and backend-specific issues that required additional debugging and configuration adjustments before successful deployment. In contrast, the same models could be directly deployed with vLLM without additional model-specific modifications or special setup.

Therefore, although SGLang delivers better performance in most of our evaluated configurations, this does not imply that it is universally preferable to vLLM. vLLM's wider adoption, broader model coverage, and more consistent out-of-the-box compatibility remain important practical advantages, which is why SPECSYS continues to support and build upon vLLM.

K EXPERIMENTAL HYPERPARAMETERS

This section summarizes the major hyperparameters and software configurations used in our experiments. Unless otherwise specified, all compared methods use the same workload replay settings, model-generation parameters, and serving configurations under the same hardware setting.

Agent Trace Collection. We collect 1,000 execution trajectories from different agent datasets and agent tasks (details in Appendix G). Each trajectory records the input and output of every LLM invocation, together with the corresponding timestamps and complete request and response contents. We split the collected trajectories into training and test sets at a ratio of 5:1. For evaluating remaining-step prediction accuracy, only trajectories in the training set are used to construct the historical profiles, while prediction is performed exclusively on trajectories from the test set. We follow the same separation in the end-to-end experiments, ensuring that no test trajectory is used for profiling and thus avoiding data leakage.

Agent Trace Replay. We replay these collected trajectories during evaluation to emulate the execution behavior of real agents without repeatedly invoking the original agent environments. For each trace-driven experiment, we replay 4,096 agent tasks by cyclically reusing trajectories from the test set. The timestamps from the online inference trace are used to determine the task arrival times, with the beginning of the selected trace segment treated as the replay start time. To maintain workload diversity, trajectories from different task categories are interleaved during replay while preserving the temporal characteristics of the original inference trace. For each experimental configuration, all compared methods use the same trajectory sequence and task arrival timestamps. Any randomized components in workload construction use a fixed random seed to avoid performance differences caused by workload sampling.

Inference Engine Configuration. The GPU configuration and tensor-parallelism (TP) degree follow the model–hardware configurations described in the main experiments. For all inference engines, we set the GPU memory utilization to 0.85, the maximum model context length to 20,480 tokens, and `max_num_seqs` to 256. Prefix caching and chunked prefill are enabled.

LLM Generation Configuration. All LLM requests use a temperature of 0, corresponding to greedy decoding, and the maximum number of generated tokens is set to 1,024. For Qwen models, we additionally disable the built-in thinking mode by setting `"chat_template_kwargs": {"enable_thinking": False}`. This setting ensures that the evaluated requests follow the same non-thinking generation mode across experiments.

ThunderAgent Configuration. For ThunderAgent, we set both `metrics-interval` and `scheduler-interval` to 5.0 seconds. Accordingly, ThunderAgent periodically collects serving statistics and performs scheduling decisions at 5-second intervals.

Speculative Decoding and Switching Configuration. We use EAGLE-3 (Li et al., 2025) as the speculative decoding method, with a default draft length of 3 unless otherwise specified. For vLLM Dynamic SD and SGLang Adaptive SD, the switching threshold T is determined separately for each model–hardware configuration from the profiling results in Appendix B. SD is enabled with a draft length of 3 when the running batch size is at or below T, and disabled by setting the draft length to 0 when the batch size exceeds T:

$$[1, T] \to 3, \qquad [T+1, 256] \to 0.$$

Here, T denotes the profiled SD break-even threshold. SPECSYS uses the same profiled threshold to determine when to switch directly between its native SD and non-SD execution paths. The thresholds for individual configurations are reported in Appendix B.

Software Environment. Table 10 lists the major software and system versions used in our experiments. Python versions used by the major agent and inference frameworks are summarized separately in Table 11.

Table 10: Software and system versions used in the experiments.

Component	Version
NVIDIA Driver	595.91.07
CUDA	13.2
ThunderAgent	0.1.0
qwen-agent	0.0.34
AgentFlow	0.1.0
SPECSYS vLLM base	v0.17.1
SPECSYS SGLang base	v0.5.14
vLLM Dynamic SD	v0.24.0
SGLang Adaptive SD	v0.5.14

Table 11: Python versions used by different agent and inference frameworks.

Framework	Python Version
qwen-agent	3.10
ThunderAgent	3.10
AgentFlow	3.11
vLLM	3.12
SGLang	3.12