ONE STATIC SCHEDULE DOES NOT FIT ALL: PLATFORM-AWARE AND RUNTIME-ADAPTIVE SCHEDULING FOR AGENTIC RL ROLLOUTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic reinforcement learning (RL) is increasingly important for advancing large language models, yet synchronous rollouts suffer from severe long-tail latency and poor resource utilization. Existing systems mitigate this bottleneck with predict-then-schedule policies that estimate trajectory durations and prioritize likely stragglers. However, two challenges limit their effectiveness. First, the optimal schedule is platform-dependent: LLM request latency depends on continuous-batch composition, while tool operations contend for parallel executors. Second, token-count and tool-latency predictions inevitably deviate from realized execution, potentially making a static schedule ineffective or even harmful.

We present *STARS*, a simulator-driven framework for platform-aware and adaptive rollout scheduling. Positioned between the agentic RL framework and execution engines, *STARS* controls LLM admission and tool dispatch without modifying the training algorithm or underlying engines. *STARS* profiles the target LLM engine and tool pools offline, then uses simulation-based planning to evaluate the counterfactual effects of candidate scheduling decisions and generate a platform-specialized ranked plan for each rollout batch. The plan accounts for composition-dependent LLM latency and tool-executor parallelism. At runtime, an impact-triggered repair mechanism detects consequential deviations and incrementally reschedules the uncommitted portion. Experiments across diverse platform configurations demonstrate an average $1.12\times$ speedup in rollout throughput over state-of-the-art synchronous RL systems. The code is temporarily accessible at `https://anonymous.4open.science/r/STARS-91FB`.

1 INTRODUCTION

LLM agents demonstrate remarkable capabilities in solving complex tasks. Agentic reinforcement learning (RL) (Zhang et al., 2026; Gao et al., 2025b) has emerged as a powerful paradigm for training such agents to perform multi-step reasoning, invoke external tools, and make adaptive decisions. Training alternates between rollout, which generates trajectories, and policy optimization, which updates model parameters. As agentic tasks grow more complex, rollout increasingly dominates training time, accounting for over 80% of end-to-end time in some systems (Zhang et al., 2026). Generated-token counts and tool latencies both exhibit heavy-tailed distributions (Qin et al., 2025; Zhou et al., 2025), causing a small fraction of trajectories to finish substantially later than the rest. Because synchronous on-policy training[1] cannot update the policy until the entire rollout batch completes, these stragglers determine iteration makespan and leave most resources idle near the barrier.

Recent work addresses this bottleneck through predict-then-schedule (He et al., 2025; Qin et al., 2025; Zhang et al., 2026): it estimates trajectory durations from historical statistics or lightweight models, then prioritizes trajectories (e.g., by longest remaining time) to reduce rollout makespan. However, this approach faces two key challenges. **First**, an agentic trajectory is not a monolithic job but a sequence of interdependent LLM and tool operations. Its completion time depends on the target platform's continuous-batching dynamics (Kwon et al., 2023) and tool-executor parallelism.

[1]We focus on synchronous on-policy training, where the rollout barrier directly exposes the long tail. The same principles extend to batched asynchronous settings, where barriers are relaxed but not eliminated.

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.

The best admission order for the same batch can therefore vary across platforms (Li et al., 2026), undermining platform-agnostic priorities. **Second**, predicted token counts and tool latencies inevitably differ from their realized values. When these errors alter resource contention or delay a critical stage, a static schedule can become stale and amplify rather than mitigate the long tail.

To address these challenges, we propose *STARS*, a **S**imulator-driven framework for **T**arget-platform-aware and Runtime-**A**daptive **R**ollout **S**cheduling. *STARS* sits between the agentic RL framework and execution engines. Through a policy-independent interface, it controls only the admission order of LLM inference and tool operations, leaving the training algorithm and underlying engines unchanged. Because each admission can have nonlinear effects on LLM batch latency and downstream tool queues, *STARS* uses simulation to evaluate the counterfactual impact of candidate decisions. It profiles the target LLM engine and tool pools offline, then uses these profiles online to generate a platform-specialized ranked plan for each rollout batch, accounting for composition-dependent LLM latency and tool-executor parallelism. During execution, *STARS* monitors progress and triggers replanning when deviations may substantially alter the critical path or cross-resource contention. Rather than rebuilding the entire plan, it incrementally repairs only the uncommitted suffix, limiting online scheduling overhead.

We instantiate *STARS* with a concrete scheduling policy. During planning, its primary heuristic prioritizes ready operations by predicted remaining critical-path time across LLM and tool stages. To capture continuous-batching interference, *STARS* simulates admitting or deferring each LLM candidate over a bounded token horizon, adjusting priorities for batch externalities to balance straggler progress against active-batch slowdown. At runtime, an event filter detects substantial deviations, while a plan-consistency test evaluates their impact. When necessary, *STARS* reruns the policy from a live snapshot with updated predictions, using priority disagreements with the active plan to guide refinement. It preserves unaffected orderings and applies repairs only when residual-rollout simulation predicts sufficient makespan reduction.

- We identify two limitations of existing agentic rollout schedulers and propose *STARS*, a platform-aware, adaptive scheduler. Across platform configurations with sufficient tool capacity to avoid severe LLM idling, *STARS* achieves $1.12\times$ the rollout throughput of state-of-the-art baselines on average.
- We present a simulator-driven framework that models continuous LLM batching and parallel tool execution to support platform-aware planning and runtime rescheduling without modifying the training algorithm or execution engines.
- We develop a scheduling policy that balances downstream critical-path urgency against each LLM admission's batching externality through bounded counterfactual simulation. An impact-triggered repair mechanism detects execution deviations and incrementally refines the uncommitted plan with negligible online overhead.

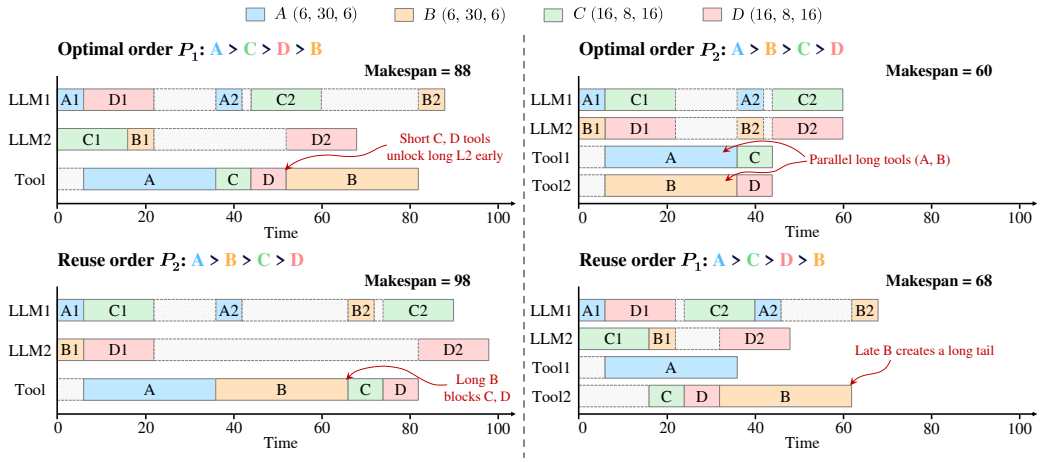
2 Background and Motivation

2.1 Agentic RL

RL post-training alternates between rollout, during which the actor samples responses, and training, which updates model parameters. Rollout is frequently the dominant bottleneck in reasoning workloads (Qin et al., 2025; Gao et al., 2025a; Zhou et al., 2025).

To preserve on-policy learning, RL post-training commonly executes these phases synchronously. All trajectories sampled by the current actor must complete before training begins; the updated weights are then propagated before the next rollout. This barrier keeps rollout data fresh and limits off-policy bias, but forces training to wait for the slowest trajectory, causing substantial tail latency, pipeline bubbles, and low accelerator utilization.

Agentic RL further exacerbates this bottleneck (Team et al., 2025). Rather than producing a single response, each trajectory alternates among LLM inference, external tool execution, and environment feedback across multiple rounds. Variability in tool latency, returned context, failures, and trajectory-dependent control flow lengthens completion times and makes them less predictable (Gao et al., 2025b). Consequently, a few stragglers can dominate rollout makespan and further degrade end-to-end training efficiency.



(a) Platform 1: 2 LLM slots + 1 Tool instance (b) Platform 2: 2 LLM slots + 2 Tool instances

Figure 1: Example of platform-dependent rollout scheduling.

2.2 RLHF Scheduling

Early RLHF systems orchestrated the pipeline of actor, critic, reference, and reward models. DeepSpeed-Chat unified generation and training through its Hybrid Engine (Yao et al., 2023), while FlexRLHF explored interleaved and disaggregated placement (Xiao et al., 2023). ReaLHF searched dataflow execution plans with parameter reallocation (Mei et al., 2024); HybridFlow combined hybrid controllers with efficient model resharding (Sheng et al., 2024). RLHFuse decomposed generation and inference into sample-level subtasks and training into microbatches, enabling inter- and intra-stage fusion (Zhong et al., 2024).

Request- and trajectory-level scheduling addresses the rollout tail more directly. RhymeRL and Seer exploit historical generation patterns (He et al., 2025; Qin et al., 2025); RollPacker concentrates tail-prone prompts into designated rollout steps (Gao et al., 2025a). APRIL resumes partial rollouts, and TailSieve uses partial rollouts to guide tail routing (Zhou et al., 2025; Xu et al., 2026). AReaL overlaps generation and training while controlling policy staleness (Fu et al., 2025). Heddle optimizes trajectory scheduling, placement, and model parallelism (Zhang et al., 2026), while RollArt disaggregates multi-task agentic training across heterogeneous resources (Gao et al., 2025b).

2.3 Challenges

2.3.1 Optimal rollout schedules are platform-dependent

Many rollout schedulers reorder trajectories using predicted response lengths or remaining execution times. Although effective in mitigating long-tail imbalance, they often treat trajectory priority as intrinsic. This assumption is inadequate for agentic RL: the optimal admission order depends on the target platform. Each trajectory alternates between LLM generation and external tool execution, forming a multistage pipeline across distinct resources. Its scheduling impact therefore depends not only on its own duration but also on how its downstream stages contend and overlap with those of other trajectories under the available capacities.

Figure 1 illustrates this dependence with four trajectories, each following $\text{LLM}_1 \rightarrow \text{Tool} \rightarrow \text{LLM}_2$. Trajectories A and B are tool-heavy, with stage durations $(6, 30, 6)$, whereas C and D are LLM-heavy, with durations $(16, 8, 16)$. Both platforms provide two LLM slots; Platform 1 has one tool instance ($T = 1$), while Platform 2 has two ($T = 2$). A fixed trajectory order determines which ready stage is dispatched whenever an executor becomes available.

On Platform 1, an optimal order is $P_1 : A \succ C \succ D \succ B$, with a makespan of 88. While A's long tool call runs, C and D complete their first LLM stages. Their short tool calls precede B's long call, allowing their long L_2 stages to overlap with B's tool execution. In contrast, $P_2 : A \succ B \succ$

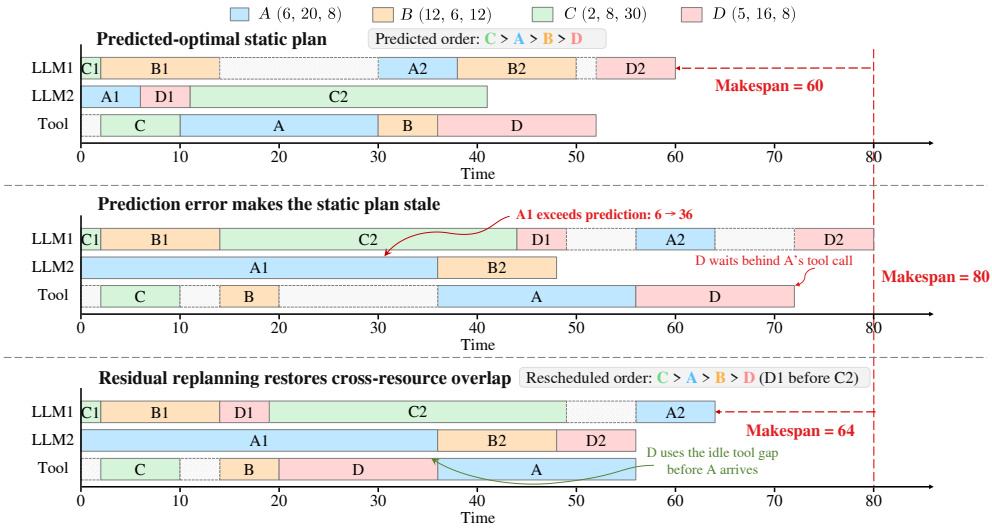


Figure 2: Example of how a prediction error degrades static scheduling performance.

$C \succ D$ runs the two long tool calls consecutively, blocking C and D, delaying their L_2 stages, and increasing the makespan to 98, an 11.4% overhead.

The ranking reverses on Platform 2. Under P_2, A and B execute their long tool calls concurrently; C and D then execute their short tool calls and L_2 stages in parallel, yielding a makespan of 60. Reusing P_1, however, postpones B's long tool call until time 32, creating a late tail and increasing the makespan to 68, a 13.3% overhead:

$$M(P_1, C_T{=}1) = 88 < 98 = M(P_2, C_T{=}1), \qquad M(P_1, C_T{=}2) = 68 > 60 = M(P_2, C_T{=}2).$$

Thus, response-length, tool-latency, and remaining-work estimates alone cannot determine a universally optimal trajectory order. Effective scheduling must jointly account for trajectory structure, cross-resource interference, and platform state.

2.3.2 Prediction errors can undermine static schedules

Prediction-based static schedules depend on stage-time estimates that may not hold during execution. In agentic RL, stochastic decoding can produce unexpectedly long responses, while tool failures, retries, and variable outputs can change tool latency and subsequent LLM-stage lengths. Because trajectories repeatedly alternate between LLM and tool resources, even a local error can shift downstream arrivals, disrupt planned overlap, and invalidate a previously optimized order.

Figure 2 illustrates this effect on a platform with two LLM slots and one tool executor. Four trajectories follow $\text{LLM}_1 \to \text{Tool} \to \text{LLM}_2$. Given predicted stage times $A = (6, 20, 8)$, $B = (12, 6, 12)$, $C = (2, 8, 30)$, and $D = (5, 16, 8)$, the scheduler derives the static order $C \succ A \succ B \succ D$. Predictions are accurate for B-D, but A_1 takes 36 rather than 6 time units. When an LLM slot becomes available at time 14, the stale order still admits C_2 before D_1. Consequently, D_1 starts only at time 44, and its tool call waits behind A's long call from time 36 to 56. The final D_2 stage therefore finishes at time 80. Across all 4! fixed trajectory orders, the average makespan is 70.5; the prediction-derived plan is thus 13.5% slower than a uniformly random order.

Runtime feedback provides an opportunity to recover. At time 6, the scheduler observes that A_1 remains unfinished and replans at the next safe admission boundary, time 14. Completed operations remain fixed; only the uncommitted plan is re-optimized using the updated state and workload estimates. The revised plan admits D_1 before C_2, allowing D's tool call to run from time 20 to 36 and fill an otherwise idle tool interval before A arrives at time 36. Meanwhile, C_2 overlaps with A's tool execution. The rollout finishes at time 64, reducing makespan by 20.0% relative to static execution.

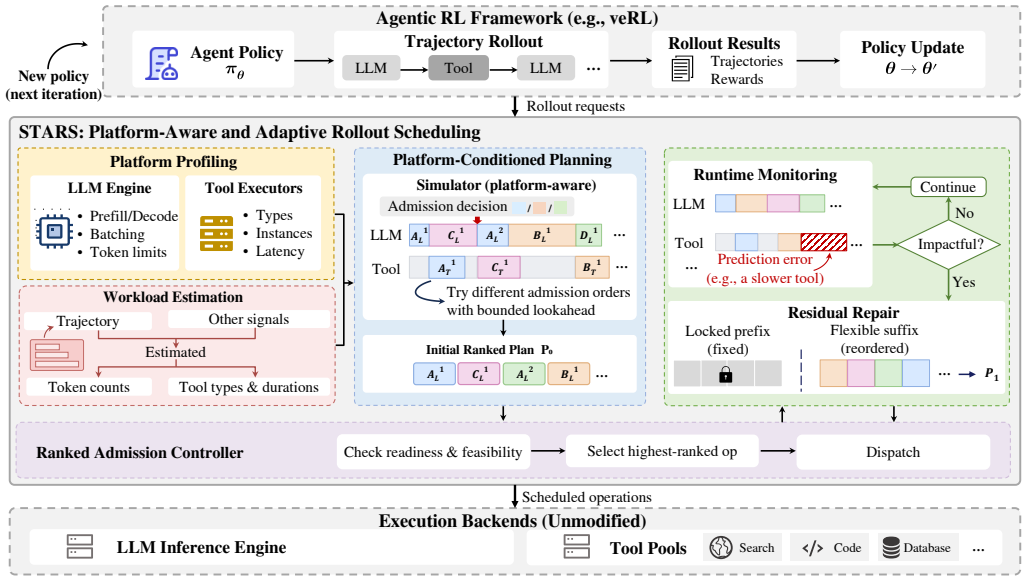


Figure 3: Overview of the *STARS* framework.

This example motivates monitoring the impact of prediction errors. Deviations that change downstream arrivals or resource competition can require a different admission order.

3 Scheduling Framework

3.1 Architecture and Scheduling Scope

As shown in Figure 3, *STARS* introduces an independent trajectory scheduling layer between an agentic RL framework's rollout orchestration layer and the underlying execution engines. The rollout controller or AgentLoop maintains the sampled request set, while the LLM engine and tool executors perform the actual computation. *STARS* preserves the framework's training algorithm and sampled requests, as well as the LLM backend's continuous batching and internal organization of prefill and decode. Instead, it determines which ready LLM operations enter the engine's waiting queue and their relative admission priorities. Similarly, it orders ready tool calls for dispatch to the corresponding executor pools. This separation enables integration with different agentic RL systems and tool runtimes without modifying their internal implementations.

3.2 Platform Profiling and Workload Estimation

STARS profiles the target platform offline to parameterize its rollout simulator. The platform model captures LLM latency, sequence and token budgets, and tool-pool capacities. For the LLM engine, profiling measures token-step latency across mixed prefill/decode batches. Tools are grouped by resource type, with each group backed by independent executors. Profiling is off the rollout critical path and repeated only after hardware, engine, or major configuration changes.

Before each rollout, an online workload estimator predicts the remaining execution structure of incoming trajectories from available information. Depending on the estimator, inputs may include historical traces, model hidden states, or other statistical signals. Its interface decouples estimation from scheduling, enabling alternative estimators to be evaluated independently. Specifically, it predicts token counts for LLM operations and execution times for tool calls. The simulator combines these predictions with the target-platform model to estimate batch-dependent LLM runtimes and tool-queuing delays under executor contention.

3.3 PLATFORM-CONDITIONED PLANNING

The planner takes the fixed set of rollout requests, workload predictions, and platform profiles, and evaluates candidate plans with the rollout simulator. It provides common plan-generation interfaces, enabling heuristic, search-based, and learned schedulers to use the same framework. The instantiated policy minimizes predicted rollout makespan, while runtime feedback revises decisions as workload estimates evolve. By evaluating each plan under the target configuration, the simulator captures configuration-dependent behavior: the same workload may induce different schedules under different resource configurations.

Rather than a timetable with fixed start times, the planner outputs ranked scheduling priorities. At each scheduling point, the runtime selects a dependency-ready, resource-feasible operation according to its LLM rank or tool score. Skipped operations are reconsidered when dependencies are resolved or resources become available. Encoding relative order instead of exact timing accommodates routine variation in generation length and tool latency while preserving trajectory dependencies.

3.4 RUNTIME MONITORING AND RESIDUAL REPLANNING

During execution, a runtime monitor compares observed progress with planning assumptions. Deviations in decode completion time, tool execution time, or tool-response prompt length may trigger replanning if they exceed the replanning threshold and structurally alter the resource state and critical path. Upon a trigger, *STARS* obtains a checkpoint from the executors. Completed operations remain fixed. The estimator updates the remaining workload, and the planner starts from the checkpoint to optimize only the flexible residual plan. Before activation, *STARS* validates the revised plan against the latest execution state and weighs its predicted benefit against replanning and switching overheads. A valid plan offering sufficient net benefit is installed at a safe admission boundary.

4 SIMULATOR-DRIVEN SCHEDULING POLICY

This section presents the concrete platform-aware scheduling and replanning policy used by *STARS*.

4.1 BATCH-AWARE CRITICAL-PATH SCHEDULING

At each scheduling opportunity, *STARS* considers ready operations satisfying current resource constraints: sequence and token budgets for LLMs, and executor availability for tools. It estimates each candidate's remaining critical path from predicted token counts and tool service times. Because tool calls occupy independent executors while LLM requests interact through continuous batching, *STARS* schedules them differently.

We model tool executors as independently provisioned resources (e.g., separate virtual machines) without memory over-commitment. Subject to executor availability, ready calls of each tool type are dispatched by decreasing prediction-only critical-path score. For LLM requests, however, longest-remaining-path scheduling exposes stragglers early but may delay requests already in the batch. *STARS* therefore selects the candidate c maximizing $S(c) = R(c) - \lambda_L E_L^H(c) + \lambda_W W_T(c) - \lambda_T E_T(c)$. Here, $R(c)$ is the reference remaining critical-path time, excluding the current prefill chunk's reference cost. $E_L^H(c)$ captures LLM batch externality through bounded lookahead. A baseline advances selected requests for up to H token steps while withholding waiting candidates; an admission branch admits c whenever the step budget permits. $E_L^H(c)$ sums positive step-latency increases over the baseline, weighted by the incumbent requests affected at each step.

To derive the tool-side terms, *STARS* simulates the projected tool phase together with active tool work and arrivals from running LLM rounds. $W_T(c)$ is the candidate's projected tool-phase completion time minus its arrival time and intrinsic tool critical-path duration, clipped at zero, and thus measures contention-induced waiting. $E_T(c)$ sums positive increases in other projected tool phases' completion times after inserting c. The nonnegative weights λ_L, λ_W, and λ_T balance LLM externality, tool-contention urgency, and tool externality, respectively.

The simulator repeatedly advances to the next scheduling opportunity and applies this policy until all predicted trajectories complete. It fixes LLM-operation ranks by first-admission order and retains initial tool critical-path scores, producing plan P and its predicted makespan. At runtime, these ranks

and scores are consulted only among ready, feasible operations; candidate simulation occurs only during planning. With extra pre-rollout planning budget, *STARS* may revisit close-score decisions whose outcomes affect a predicted tail trajectory.

4.2 Impact-Triggered Rescheduling with Incremental Repair

The initial plan relies on predicted decode lengths, tool durations, and tool-return prompt sizes, which may differ from realized values. Yet, *STARS* does not replan after every decode completion or minor prediction error, as most small deviations can be absorbed without affecting the rollout tail.

The runtime monitor first applies a lightweight event filter, initiating a consistency check only for substantial deviations: a decode exceeds its predicted upper bound without completing or finishes substantially earlier than expected; a tool call exceeds its expected duration; a tool response yields an unexpectedly large prompt; or effective LLM or tool capacity changes.

Passing this filter does not itself trigger replanning. *STARS* then performs a fast plan-consistency test using the updated progress of affected trajectories. First, it checks whether delays remain within these trajectories' predicted slack and are thus unlikely to affect rollout makespan. Second, it recomputes remaining-path priorities for affected trajectories and currently ready operations. The local ordering is considered stable if the operation selected by the active plan remains among the highest-priority candidates. Third, it checks whether the deviation alters a critical cross-resource relationship, such as reversing the arrival order of two tool calls, delaying a tool call that unlocks a long LLM suffix, or changing the predicted straggler.

When these checks reveal a substantial scheduling impact, *STARS* replans the residual rollout from a live snapshot, keeping completed operations fixed. Rather than rebuilding the entire suffix, it performs plan-anchored repair: updating remaining-work predictions, rerunning the base policy from the snapshot, and identifying disagreements with the active plan. Only operations on affected trajectories, competing in their tool pools, or lying on the newly predicted tail may be reordered. Repair typically overrides only one or two admission decisions. A full residual-plan rebuild is considered only after structural changes, such as capacity failure, simultaneous deviations across trajectories, or a shift in the dominant bottleneck. Each candidate plan is simulated from the same snapshot to rollout completion and adopted only if it reduces predicted remaining makespan by a sufficient margin.

5 Evaluation

This section evaluates the end-to-end performance of *STARS* and then presents breakdown analyses of platform-aware planning, runtime rescheduling, and scheduling overhead.

5.1 Experimental Setup

Infrastructure. We conduct experiments on four servers, each with eight Ascend 910B2 accelerators. Tool calls execute on a separate server with a 36-core Intel Xeon W9-3475X processor and 1 TB of memory. We denote each machine configuration by $\mathcal{C} = (L, T)$, where L is the number of LLM inference servers and T is the number of tool-executor virtual machines. We self-host instruction-tuned DeepSeek-V4-Flash models that support reasoning and tool use. We use vLLM 0.21.0 (Kwon et al., 2023) with the vLLM-Ascend plugin as the inference engine, Harbor 0.7.1 (Shaw, 2025) as the benchmark framework, and opencode 1.14.41 (Anomaly, 2025), modified only to record per-operation execution traces, as the coding agent.

Datasets. We combine multi-turn LLM-tool interaction traces from DevOpsGym (build) (Tang et al., 2026), SWE-bench Verified (Jimenez et al., 2024), Terminal-Bench 2.1 (Merrill et al., 2026), and Terminal-Bench Pro (Wang et al., 2025) to construct our training dataset.

Baselines. We compare *STARS* with veRL (Sheng et al., 2024), a state-of-the-art open-source agentic RL framework. veRL schedules tool operations in first-in-first-out order and admits the next ready LLM operation whenever inference capacity is available.

Rollout Replay. To ensure fair comparisons, we replay identical recorded trajectories across scheduling policies and resource configurations. For each task, we collect multiple runs, using

earlier runs as prediction history and the final run as the replay trace. During replay, token counts, tool service times, and dependencies remain fixed, whereas LLM batch composition, queueing, and cross-resource overlap vary with scheduling decisions. This method provides reproducible comparisons while preserving variability observed in real agent executions.

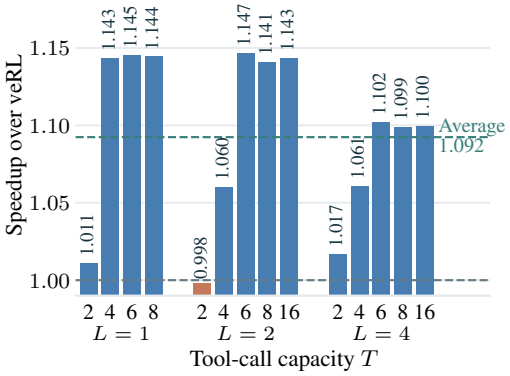


Figure 4: Rollout speedups of *STARS* over veRL.

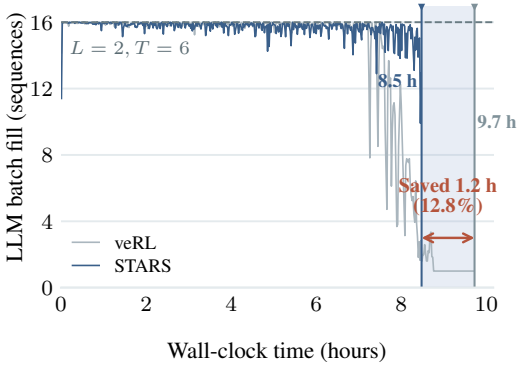


Figure 5: LLM batch fill.

5.2 Overall Performance

We compare *STARS* with veRL across resource configurations, varying the number of LLM servers $L \in \{1, 2, 4\}$ and concurrent tool-call capacity $T \in \{2, 4, 6, 8, 16\}$.

Figure 4 reports rollout speedups over veRL, computed from net makespans that include the planning overhead of *STARS*. *STARS* outperforms veRL in nearly all configurations, achieving average and maximum speedups of $1.09\times$ and $1.15\times$, respectively. Figure 5 shows the number of in-flight prefill and decode sequences in the LLM batch for $L = 2$ and $T = 6$. *STARS* sustains higher occupancy, particularly near rollout completion, shortening makespan and freeing LLM servers over one hour earlier.

Sufficient tool capacity enables *STARS* to exploit scheduling flexibility. Under severe tool constraints, tool-side queueing dominates, limiting the benefits of reordering LLM and tool operations. As detailed in Appendix B, makespan with $T = 2$ is 1.6–1.9$\times$ that with $T = 4$, highlighting the importance of avoiding severe tool bottlenecks when provisioning rollout resources. Excluding configurations with $T = 2$, the average speedup of *STARS* increases to $1.12\times$, better reflecting its practical potential.

STARS delivers greater gains when LLM capacity is limited relative to rollout batch size. With fewer LLM servers, admission order more strongly affects resource contention and downstream tool arrivals. Excluding $T = 2$, average speedup is $1.14\times$ with $L = 1$, compared with $1.09\times$ with $L = 4$. This regime is practically relevant because LLM servers require accelerators and are often provisioned more sparsely than tool executors.

5.2.1 Effectiveness of Platform-Aware Scheduling

We evaluate platform-aware planning under six resource configurations, $\mathcal{C}_i = (L_i, T_i) \in \{(1, 2), (1, 4), (2, 4), (4, 4), (4, 8), (4, 1)\}$. For each source configuration $\mathcal{C}_i$, we generate a plan P_i and transfer it to all six target configurations $\mathcal{C}_j$. We report the net makespan reduction achieved by the target-specific plan P_j over the transferred plan P_i on $\mathcal{C}_j$.

Figure 6 shows that, excluding transfers to $\mathcal{C} = (4, 1)$, target-specific plans reduce net makespan by 6.03% on average and up to 14.2% relative to transferred plans, demonstrating the benefit of platform-aware planning. In contrast, for transfers to $\mathcal{C} = (4, 1)$, the reduction is at most 0.167%. This insensitivity arises from a tool-side bottleneck: despite greater LLM batch capacity, the single tool executor serializes tool calls, leaving little scheduling flexibility.

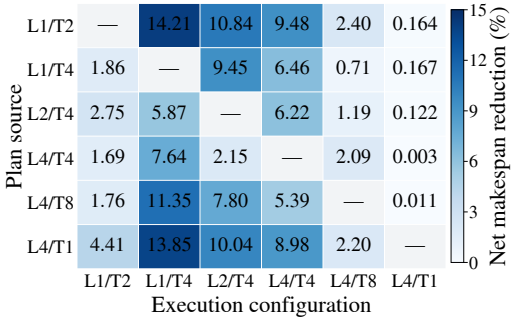


Figure 6: Net makespan reduction (%) of target-specific plans over transferred plans.

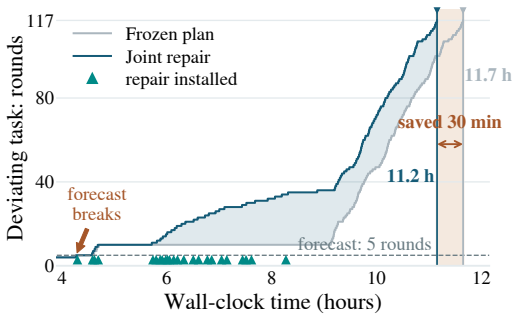


Figure 7: Runtime repairs reduce makespan under execution deviations.

5.3 Effectiveness of Runtime Rescheduling

We evaluate runtime rescheduling using a fixed forecast from the recorded base batch and a controlled deviation in the execution trace. We preserve the target trajectory's first four rounds and extend its suffix using a fixed construction from recorded traces, while keeping the other 191 trajectories unchanged.

Figure 7 shows the execution timeline for the tail-113-decode8 condition at $L = 4, T = 4$. Although numerous deviations occur, *STARS* filters out most without triggering repairs. The forecast breaks at approximately 4.27 hours, and the first accepted repair is installed shortly afterward. Several further repairs occur mid-rollout, reducing makespan by 4.26% relative to freezing the initial plan.

Repairs cluster near the middle for two reasons. First, mid-rollout deviations are more likely to change critical-path priorities and resource contention. Second, early deviations can often be absorbed by slack across numerous remaining LLM and tool operations, whereas late deviations leave too few operations and insufficient scheduling flexibility to yield meaningful improvements.

5.4 Scheduling Overhead

We evaluate the overhead of initial planning and runtime rescheduling. Initial planning takes 6.8–17.3 seconds on a host CPU (Table 4 in Appendix C), accounting for 0.02%–0.06% of rollout makespan. It can also overlap with execution of the preceding rollout batch, potentially hiding its latency entirely. Across the twelve evaluated conditions, the observed net makespan impact is at most 3.1 seconds, measured as the makespan difference between running the full repair pipeline without installing any repair and freezing the plan (Table 6 in Appendix C), less than 0.01% of overall makespan. Event and impact filters suppress unnecessary rescheduling, making overhead proportional to the number of deviations that trigger repair. When no deviation triggers repair, no rescheduling is performed. Overall, *STARS* introduces negligible overhead, which can be reduced further through execution overlap and selective rescheduling.

6 Conclusion

This paper presents *STARS*, a framework for platform-aware, adaptive rollout scheduling in agentic reinforcement learning. *STARS* combines simulator-driven planning tailored to platform resources with runtime repair to accommodate prediction errors. Its scheduling policy balances remaining-path urgency and tool waiting time against LLM and tool externalities, while its runtime rescheduling policy triggers replanning only when necessary to limit scheduling overhead. Evaluation shows that *STARS* achieves an average $1.12\times$ rollout speedup over veRL on configurations with sufficient tool capacity to avoid severe LLM idling, by mitigating stragglers and improving cross-resource overlap.

AI USE STATEMENT

We used generative AI tools to polish the writing, assist with coding and experiment execution, and perform statistical analyses of the results. In addition, the recorded trajectories that constitute our evaluation workload were generated by an LLM coding agent (Section 5.1); our experiments replay their recorded structure and timing rather than any generated text. We did not use these tools to formulate research ideas or generate entire paragraphs or sections. The authors reviewed all AI-assisted manuscript content and verified and tested AI-generated code for correctness. We take full responsibility for the final content of this work, including all text, claims, and artifacts produced with generative AI assistance.

REFERENCES

Anomaly. Opencode: The open source coding agent, 2025. URL https://github.com/anomalyco/opencode.

Wei Fu, Jiaxuan Gao, Xujie Shen, et al. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025. URL https://arxiv.org/abs/2505.24298.

Wei Gao, Yuheng Zhao, Dakai An, et al. RollPacker: Mitigating long-tail rollouts for fast, synchronous RL post-training. *arXiv preprint arXiv:2509.21009*, 2025a. URL https://arxiv.org/abs/2509.21009.

Wei Gao, Yuheng Zhao, Tianyuan Wu, et al. RollArt: Disaggregated multi-task agentic RL training at scale. *arXiv preprint arXiv:2512.22560*, 2025b. URL https://arxiv.org/abs/2512.22560.

Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History rhymes: Accelerating LLM reinforcement learning with RhymeRL. *arXiv preprint arXiv:2508.18588*, 2025. URL https://arxiv.org/abs/2508.18588.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

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

Yueying Li, Yuanfan Chen, Jiayang Chen, Esha Choukse, Haoran Qiu, G Edward Suh, Rodrigo Fonseca, Ziv Scully, and Udit Gupta. Beyond prediction: Tail-aware scheduling for llm inference. *arXiv preprint arXiv:2606.18431*, 2026.

Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. ReaLHF: Optimized RLHF training for large language models through parameter reallocation. *arXiv preprint arXiv:2406.14088*, 2024. URL https://arxiv.org/abs/2406.14088.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang,

Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Ruoyu Qin, Weiran He, Weixiao Huang, Yangkun Zhang, Yikai Zhao, Bo Pang, Xinran Xu, Yingdi Shan, Yongwei Wu, and Mingxing Zhang. Seer: Online context learning for fast synchronous LLM reinforcement learning. *arXiv preprint arXiv:2511.14617*, 2025. URL https://arxiv.org/abs/2511.14617.

A. Shaw. Harbor framework, 2025. URL https://github.com/laude-institute/harbor.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. *arXiv preprint arXiv:2409.19256*, 2024. URL https://arxiv.org/abs/2409.19256.

Yuheng Tang, Kaijie Zhu, Bonan Ruan, Chuqi Zhang, Michael Yang, Hongwei Li, Suyue Guo, Tianneng Shi, Zekun Li, Christopher Kruegel, et al. Devops-gym: Benchmarking ai agents in software devops cycle. *arXiv preprint arXiv:2601.20882*, 2026.

MiroMind Team, Song Bai, Lidong Bing, Carson Chen, Guanzheng Chen, Yuntao Chen, Zhe Chen, Ziyi Chen, Jifeng Dai, Xuan Dong, et al. Mirothinker: Pushing the performance boundaries of open-source research agents via model, context, and interactive scaling. *arXiv preprint arXiv:2511.11793*, 2025.

Weixun Wang, XiaoXiao Xu, Xander Xu, et al. Let it flow: Agentic crafting on rock and roll, building the rome model within an open agentic learning ecosystem, 2025. URL https://arxiv.org/abs/2512.24873.

Youshao Xiao, Zhenglei Zhou, Fagui Mao, Weichang Wu, Shangchun Zhao, Lin Ju, Lei Liang, Xiaolu Zhang, and Jun Zhou. An adaptive placement and parallelism framework for accelerating RLHF training. *arXiv preprint arXiv:2312.11819*, 2023. URL https://arxiv.org/abs/2312.11819.

Tianqi Xu, Lu Lv, Haoyang Huang, et al. TailSieve: Partial-rollout-guided tail routing for LLM rollouts. *arXiv preprint arXiv:2608.22788*, 2026. URL https://arxiv.org/abs/2608.22788.

Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, et al. DeepSpeed-Chat: Easy, fast and affordable RLHF training of ChatGPT-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023. URL https://arxiv.org/abs/2308.01320.

Zili Zhang, Yinmin Zhong, Chengxu Yang, Chao Jin, Bingyang Wu, Xinming Wei, Yuliang Liu, and Xin Jin. Heddle: A distributed orchestration system for agentic RL rollout. *arXiv preprint arXiv:2603.28101*, 2026. URL https://arxiv.org/abs/2603.28101.

Yinmin Zhong, Zili Zhang, Bingyang Wu, et al. Optimizing RLHF training for large language models with stage fusion. *arXiv preprint arXiv:2409.13221*, 2024. URL https://arxiv.org/abs/2409.13221.

Yuzhen Zhou, Jiajun Li, Yusheng Su, et al. APRIL: Active partial rollouts in reinforcement learning to tame long-tail generation. *arXiv preprint arXiv:2509.18521*, 2025. URL https://arxiv.org/abs/2509.18521.

A Workload Description

Workload composition. Our evaluation workload comprises recorded multi-turn agent executions from DevOpsGym (build), SWE-bench Verified, Terminal-Bench 2.1, and Terminal-Bench Pro.

Table 1: Composition of the evaluation batch by benchmark. Counts denote recorded execution trajectories.

Benchmark	Trajectories
DevOpsGym (build)	76
SWE-bench Verified	24
Terminal-Bench 2.1	43
Terminal-Bench Pro	49
Total	192

These benchmarks span software builds, repository-level bug fixing, and terminal workflows involving compilation, testing, and model training. Table 1 reports the batch composition by benchmark. Multiple executions of the same task are retained as distinct trajectories.

Trace representation and batch construction. Each trajectory records the sequence of LLM generations and tool operations, including prompt and output token counts for each LLM round, tool types, and tool service times. Replay preserves these attributes and dependencies, whereas admission order, queueing, and overlap are determined by the scheduling policy and available resources.

We construct the evaluation batch from a fixed pool of 197 trajectories. We sort them by request identifier, shuffle them with seed 20260912, and retain the first 192 without replacement. The initial-planning and platform-transfer experiments reuse this fixed batch across resource configurations and scheduling policies.

Workload characteristics. The batch contains 5,472 LLM rounds and 6,087 tool calls, totaling 174,620,692 prompt tokens and 1,527,092 output tokens. Prompt-token counts are aggregated across rounds and include conversation history repeated in successive prompts. Table 2 reports the workload distributions.

Table 2: Workload characteristics of the 192 evaluation trajectories. Each row reports statistics over trajectories, LLM rounds, or tool calls, as applicable.

Metric	Median	P90	Maximum
LLM rounds per trajectory	20	57.9	122
Output tokens per trajectory	4,491.5	19,448.9	57,844
Tool calls per trajectory	24	63.8	122
Prompt tokens per LLM round	25,234	65,277.8	103,101
Tool service time per call (s)	0.021	30.009	3,600.296

The workload exhibits long interaction chains and substantial variation in tool duration: 28 trajectories contain at least 50 LLM rounds, and 51 tool calls last at least ten minutes. Together, these characteristics create long dependency chains and tool-induced stalls, motivating joint scheduling of LLM and tool operations.

B Absolute Execution Times of veRL and *STARS*

Table 3 reports absolute net rollout makespans for veRL and *STARS* at $L = 2$ across tool capacities $T \in \{1, 2, 4, 6, 8, 16\}$. The *STARS* makespans include initial planning time. Mean decode occupancy denotes the time-averaged fraction of occupied LLM decode slots.

Makespan drops sharply as T increases to 4 and plateaus beyond $T = 6$. By $T = 6$, the tool pool accommodates the workload's intrinsic tool parallelism, and *STARS* reduces rollout time from 9.7 to 8.5 hours (1.14–1.15×). At $T \leq 2$, tool-side queueing dominates regardless of scheduling. Makespan at $T = 2$ is 1.74× (veRL) and 1.84× (*STARS*) of that at $T = 4$; at $T = 1$, it rises to about 34 hours, or 3.4–3.6× the $T = 4$ level. Mean decode occupancy for veRL falls from 0.70–0.78 at $T \geq 4$ to 0.27 at $T = 2$ and 0.11 at $T = 1$, leaving the LLM servers mostly idle while waiting for saturated tool executors. In this regime, reordering LLM admissions and tool dispatch

Table 3: Absolute net makespan and mean LLM decode occupancy at $L = 2$ across tool capacities T. Speedup is veRL makespan divided by *STARS* makespan.

	Net makespan (s)			Mean decode occupancy	
T	veRL	*STARS*	Speedup	veRL	*STARS*
1	122,413	121,299	$1.009\times$	0.11	0.12
2	62,004	62,112	$0.998\times$	0.27	0.30
4	35,697	33,673	$1.060\times$	0.70	0.78
6	34,995	30,513	$1.147\times$	0.77	0.92
8	34,730	30,449	$1.141\times$	0.78	0.92
16	34,783	30,424	$1.143\times$	0.78	0.92

cannot increase tool throughput, so *STARS* essentially matches veRL ($0.998\times$ at $T = 2$ and $1.009\times$ at $T = 1$), the only configurations without meaningful gains in the sweep. Such severely tool-bottlenecked configurations waste LLM capacity and wall-clock time and should be avoided when provisioning rollout platforms. This motivates reporting the $T = 2$-excluded average in Section 5.2 alongside the all-configuration average.

C Detailed Analysis of *STARS* Runtime Overhead

Table 4 reports *STARS*'s measured initial-planning time across configurations: 6.8–17.3 seconds per batch, at most 0.06% of the corresponding *STARS* makespan.

Table 4: Measured initial-planning time (seconds) per configuration in the overall sweep.

	T=2	T=4	T=6	T=8	T=16
$L = 1$	12.4	15.8	8.0	6.8	–
$L = 2$	11.0	16.1	12.1	9.9	10.2
$L = 4$	9.6	17.3	15.9	14.0	13.5

Rescheduling uses the per-stage fees in Table 5. Each resource configuration is calibrated using all four inputs and all three controllers; a full-stage probe provides additional coverage. Each fee equals the larger of the maximum observed stage duration and the reservation already advanced, plus 10%, rounded up to the nearest millisecond. Thus, the 30-second planning reservation yields a 33-second frozen planning fee.

Table 5: Frozen fees per control-stage invocation (seconds) at $T = 4$.

L	Initial	Checkpoint	Monitor	Planning	Validation	Install
1	17.437	0.037	0.075	33.000	0.688	0.036
2	17.944	0.034	0.076	33.000	0.682	0.036
4	19.335	0.068	0.104	33.000	1.316	0.036

At most two additional retries are allowed per structural-failure chain, subject to the global 64-attempt limit. Execution continues throughout the planning reservation. Validation uses a window equal to twice the sum of the validation and installation fees, temporarily withholding new LLM admissions while in-flight work continues. Candidates are rechecked for benefit and exact state agreement before installation; a planning overrun prevents installation. Frozen fees define the applied cost model, while actual stage durations are recorded separately.

Table 6 reports, for each condition, repair attempts and accepted installations, total fees charged across five stages, measured host-side durations for those stages, and observed net makespan impact, defined as No-install minus Frozen. Because charged and measured stage times overlap with in-flight execution, they do not add directly to makespan. Across all twelve conditions, net impact

Table 6: Runtime rescheduling costs at $T = 4$.

L	Input	Attempts	Accepted	Charged (s)	Measured (s)	Net (s)
1	zero-error	0	0	784.3	73.6	0.00
1	tail-32	14	12	1,249.8	96.6	−0.67
1	tail-113	14	12	1,263.2	108.8	−0.67
1	tail-113-decode8	5	3	964.8	92.4	+2.32
2	zero-error	0	0	775.0	74.3	0.00
2	tail-32	18	12	1,370.8	105.2	+2.42
2	tail-113	18	12	1,384.0	117.3	+2.42
2	tail-113-decode8	6	3	987.9	95.7	+3.04
4	zero-error	0	0	1,177.7	77.1	0.00
4	tail-32	42	23	2,598.7	290.8	+1.85
4	tail-113	64	35	3,344.3	524.0	+1.85
4	tail-113-decode8	64	24	3,335.2	295.3	−2.08

remains within -2.1 to $+3.1$ seconds, at most 0.01% of makespan, and is negative in some cases. This comparison includes trigger-sequence changes caused by suppressing installation.