SCHEDULING KERNEL-GENERATING AGENTS AT SCALE IN DECOUPLED AGENT–WORKER SYSTEMS

Anonymous authors
Paper under double-blind review

Abstract

Programming agents develop GPU kernels by repeatedly generating, compiling, and running code. In production, the large volume of operator-generation tasks motivates a decoupled Agent–Worker architecture that avoids dedicating GPUs to individual agents during model inference and tool use. Agents run on CPU clusters and submit code under test to a shared GPU execution pool when compilation and testing are needed. In this setting, each batch's Skill, the types and number of operators to generate, and model inference speed jointly determine how frequently the batch requires GPU execution. Too few execution requests leave GPUs idle, whereas too many cause queues to accumulate, delaying execution feedback and reducing useful iterations, which can compromise kernel quality.

We propose a Scheduling Agent that controls the concurrency and admission of operator-generation tasks. It reads each batch's Skill to identify the stages that require GPUs, then combines resource utilization, the queue of code awaiting execution, and model-API feedback to select the next tasks to launch. It also uses file semantics to promptly identify and remove residual artifacts created at incorrect output paths, reclaiming disk space and inodes. Compared with FIFO alone, adding the Scheduling Agent increases Worker utilization from 59.9% to 85.2%. It also removes approximately 2.2 TB of residual files from incorrect paths and frees about 380,000 inodes. The system has operated in production for several months, providing scheduling support for over ten thousand operator-generation tasks each month.

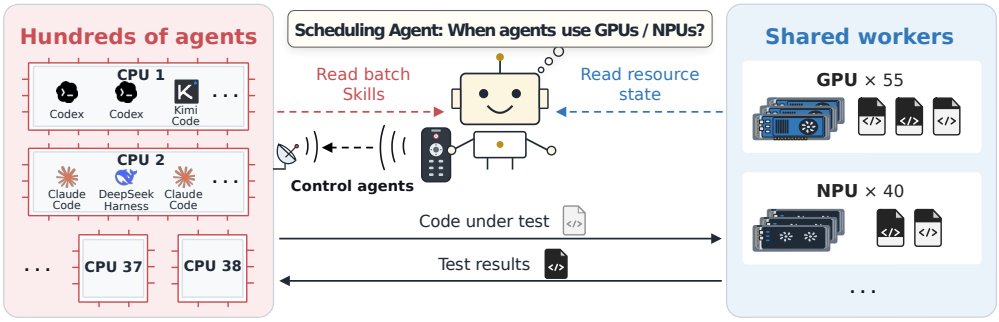


Figure 1: System overview. Programming agents generate code on a CPU cluster and submit code under test to a shared Worker pool comprising different hardware types, including GPUs and NPUs. The Scheduling Agent reads batch Skills and resource states to control agent concurrency.

1 INTRODUCTION

Advances in the ability of large language models and programming agents to write code, invoke tools, and sustain long-horizon iteration are making the automation of complex software development increasingly feasible. Operator generation exemplifies this trend: agents can read specifications and documentation, invoke compilation, testing, and profiling tools, and optimize code based on execution feedback. For example, KernelBench studies iterative GPU kernel generation using

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.

correctness and performance feedback (Ouyang et al., 2025). SOL-ExecBench derives GPU kernel optimization tasks from real cluster workloads and evaluates generated kernels against analytically derived hardware performance limits (Lin et al., 2026). In production, however, operator development must cover basic operators in neural-network and linear-algebra libraries, fused operators such as FlashAttention (Dao et al., 2022), and specialized operators required by inference frameworks such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024). Each operator must also support different hardware versions, programming approaches such as CUDA and Triton, and a range of tensor shapes and data types, resulting in a substantial volume of generation work.

As illustrated in Figure 1, production deployments commonly support large-scale operator generation by decoupling programming agents running on CPU clusters from shared GPU Workers. Agents submit code under test and retrieve compilation, testing, and profiling results, allowing multiple tasks to share GPUs through interleaved execution. In this setting, a CPU host can support dozens of agents, whereas each GPU is configured to execute one submitted test workload at a time, making GPU capacity a more constrained resource than CPU capacity. GPU Worker utilization therefore depends on how frequently agents submit code under test: too few submissions leave devices idle, whereas submissions that exceed Worker capacity cause queueing, delay feedback, and reduce opportunities for validation and optimization. The system must therefore dynamically coordinate task supply to balance resource utilization and timely feedback.

The scheduling challenge is that equal task counts do not imply equal GPU demand. As shown in Figure 2, even with the same task count, model, and other settings, changing the Skill used for operator generation can substantially alter both the number and timing of submissions to shared GPU Workers. These differences arise because Skills prescribe different operator-generation workflows.

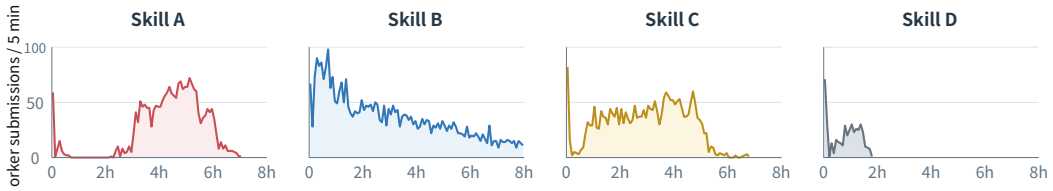


Figure 2: Distribution of code-under-test submissions to shared GPU Workers under different Skill configurations. The four historical batches have the same task count and model configuration. The horizontal axis shows elapsed time since task pickup, and the vertical axis shows the number of code-under-test submission events in each five-minute window.

Some emphasize parameter search and performance testing, producing frequent submissions early in execution. Others devote substantial initial time to developing textual implementation plans, delaying the production of code ready for testing. Task concurrency must therefore be determined not only by the number of GPUs, but also by the development workflow prescribed by each Skill and the progress of the tasks. In addition, model-API inference speed, rate limits, and quota exhaustion can change the supply of code under test.

To address these challenges, we propose a scheduling approach that controls task supply in the Agent–Worker architecture. The Scheduling Agent reads task Skills, configurations, runtime states, and historical records, and combines this information with GPU Worker resource states and model-API service feedback to dynamically adjust the concurrency and admission of operator-generation tasks. It also uses semantic analysis and task states to identify intermediate files created at incorrect output paths and reclaim the corresponding disk space and inodes. This addresses the accumulation of residual files during long-running operation, where inode usage can grow faster than disk-space usage.

We evaluate the approach in a real, large-scale operator-generation deployment that has operated continuously for several months, collecting extensive operational data and covering representative issues encountered in practice. We compare scheduling augmented with the Scheduling Agent against the system's existing first-in, first-out (FIFO) baseline. FIFO dispatches tasks in arrival order and waits when the task at the head of the queue cannot currently execute. The Scheduling Agent augments this baseline by adjusting concurrency and admission based on task Skills, configurations, and runtime states. The results show that adding the Scheduling Agent improves GPU Worker uti-

lization relative to FIFO and enables timely adjustments to the concurrency of affected tasks when model-API service conditions fluctuate. Its semantic analysis and cleanup mechanism also mitigates the accumulation of residual files during long-running operation.

2 Related Work

Programming agents. ReAct interleaves language-model reasoning with tool use, while Toolformer uses self-supervised learning to learn when to invoke external tools (Yao et al., 2023; Schick et al., 2023). Building on these capabilities, AutoGen and OpenHands organize multi-agent conversations, code execution, and environment interaction into extensible agent frameworks (Wu et al., 2024; Wang et al., 2025). SWE-agent further introduces an Agent–Computer Interface tailored to terminals and code repositories, enabling agents to search files, run tests, and submit patches (Yang et al., 2024). Real-world software engineering capabilities are commonly evaluated in benchmark environments: SWE-bench derives issue-resolution tasks from GitHub repositories, AgentBench covers diverse interactive tasks, and WebArena provides browser environments with realistic website states (Jimenez et al., 2024; Liu et al., 2024; Zhou et al., 2024). Terminal-Bench extends this evaluation to challenging terminal tasks involving complete command-line environments, toolchains, and test-based verification (Merrill et al., 2026).

Kernel generation. The use of test feedback in code generation has precedents in CodeT, which selects candidate programs using generated tests (Chen et al., 2023). KernelBench provides an execution-based evaluation of GPU kernel generation that considers both functional correctness and performance, while AlphaEvolve iteratively improves algorithmic code through evolutionary search and external evaluators (Ouyang et al., 2025; Novikov et al., 2025). In SOL-ExecBench, NVIDIA introduces SOLAR (SOL Analysis for Runtime), which estimates theoretical lower bounds on operator execution time from computational work, data movement, and peak hardware compute throughput and bandwidth. These bounds provide a reference for assessing the remaining optimization potential of generated kernels (Lin et al., 2026). CUDA Agent combines large-scale agentic reinforcement learning, a CUDA Skill, automated verification, and profiling into a training system whose environment separates CPU-side development terminals from a GPU testing pool (Dai et al., 2026). KernelEvolve extends agentic kernel development to heterogeneous accelerators, using graph search, selection strategies, and retrieval-augmented prompts to optimize kernels across hardware platforms and programming abstractions (Liao et al., 2025). CuTeGen organizes generation, testing, and refinement into an agent workflow for CuTe, deferring low-level performance feedback to avoid premature optimization (Saba et al., 2026). These systems reflect a recent shift from general-purpose code generation toward specialized, verifiable GPU kernel generation.

Semantics-driven scheduling and resource control. Parrot represents data dependencies between LLM requests using semantic variables and exploits them to optimize application execution. AIOS abstracts agents' access to models, tools, and storage into manageable system resources (Lin et al., 2024; Mei et al., 2025). Autellix treats agent programs comprising multiple LLM calls as first-class scheduling entities and uses program-level context to mitigate head-of-line blocking across calls. Throughput-Optimal Scheduling and JITServe develop scheduling policies targeting throughput optimality and SLO constraints under incomplete request information, respectively (Luo et al., 2025; Dai et al., 2025; Zhang et al., 2025). For GPU serving in agent workflows, SAGA treats an entire workflow as an atomic scheduling unit and uses execution graphs, session-affine batching, and work stealing to reduce state-management overhead across calls (Guo et al., 2026). At the operating-system level, SchedCP separates workload analysis, policy synthesis, and execution validation to generate deployable eBPF scheduling policies for Linux (Zheng et al., 2025). These approaches primarily schedule model calls, tool executions, or workflow stages that have already been instantiated. Our work focuses on controlling admission on the supply side, before programming agents submit code under test. Skills and model-API feedback affect the number and timing of subsequent execution requests, motivating the retrieval of comparable historical runs, prediction of GPU usage profiles for candidate batches, and decisions about whether to admit or defer each batch.

3 SEMANTICS-AWARE TASK SCHEDULING

We formulate task scheduling as an admission problem: given the capacity of a Worker pool and a set of waiting batches, determine which batches to run, when to run them, and how much concurrency to allocate to each batch, without automatically expanding the resource pool. The Scheduling Agent periodically reads the current queue, Worker states, task configurations, and model-service events. It uses historical execution records to estimate the resource demand of candidate batches and proposes admission decisions or concurrency adjustments. The objective is to increase the number of tasks completed by their deadlines under a fixed workload and budget, while limiting queueing delays and resource idleness.

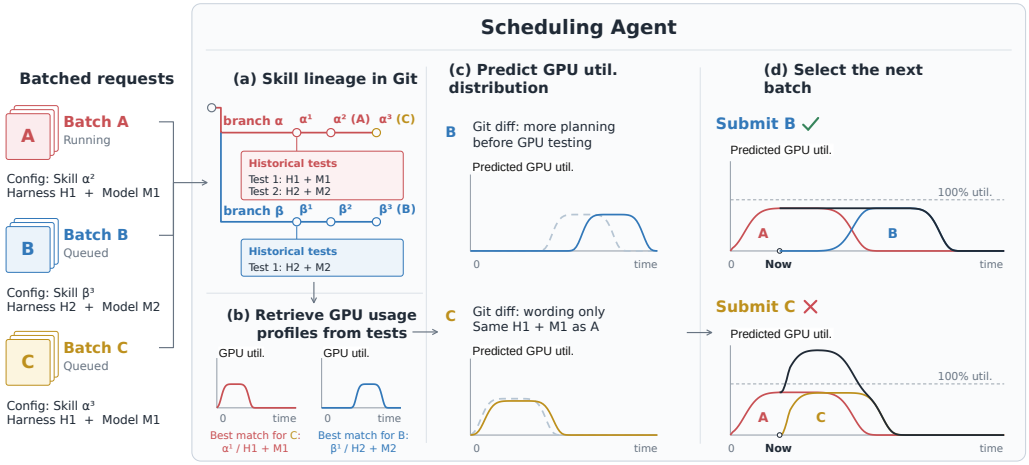


Figure 3: The four stages of semantics-aware task scheduling. (a) Trace the Skill's Git lineage to identify the commit history of the current branch. (b) Retrieve GPU usage profiles from comparable historical runs. (c) Convert the retrieved profiles into predicted GPU capacity demand for candidate batches. (d) Select the next batch to admit under capacity constraints.

3.1 SKILL LINEAGE AND EXECUTION CONFIGURATION

In Figure 3(a), the scheduler traces the Skill's Git lineage to identify the commit history of the current branch. It then compares batches jointly in terms of their Skills, models and Harnesses, task composition, and execution environments. A Skill specifies the agent's execution stages, tools, and stopping conditions; changes to these specifications can alter the timing of code-under-test submissions. At the beginning of each scheduling round, the system freezes a configuration snapshot for each candidate batch b:

$$\chi_b = (s_b, \gamma_b, w_b, e_b), \tag{1}$$

where s_b specifies the Skill version and stage order, γ_b is the Model–Harness configuration, w_b describes the task count and operator composition, and e_b specifies the devices and toolchain.

Differences between Skill commits serve only as semantic cues. They are used to calibrate historical demand only when the Model–Harness configuration, task count, and execution environment are comparable; otherwise, they are used solely to rank retrieval candidates.

3.2 RETRIEVING GPU USAGE PROFILES FROM HISTORICAL RUNS

Figure 3(b) illustrates the retrieval of historical execution records. This step constructs an auditable reference set based on configuration criteria, rather than relying exclusively on a single most similar batch. Each historical record $r \in \mathcal{D}_t$ contains its configuration, code-under-test submission trace, observation window, and execution outcomes. The scheduler first filters out records with incompatible devices, environments, task counts, operator compositions, or Model–Harness configurations.

Among the remaining records, the scheduler prioritizes direct references with matching Skills, Model–Harness configurations, task counts, and environments. If these records are insufficient, it relaxes the Skill-version requirement while keeping the Model–Harness configuration, task count, and environment unchanged, and checks the Skill's stage order, validation constraints, and tool dependencies.

At each tier, only records that pass completeness checks are retained, and at least K records are required for the tier to be eligible. Denoting the resulting candidate set by $\mathcal{V}_t^{(\ell)}(b)$, we select

$$\ell^*(b) = \min\left\{\ell \in \{1,2,3\} : \left|\mathcal{V}_t^{(\ell)}(b)\right| \geq K,\ \text{valid} = 1\right\}, \qquad \mathcal{N}_b = \mathrm{TopK}\left(\mathcal{V}_t^{(\ell^*(b))}(b)\right). \quad (2)$$

3.3 Predicting GPU Usage Profiles

Figure 3(c) converts historical GPU usage profiles into demand predictions for candidate batches. Let $\mathcal{N}_b$ denote the selected reference set. The scheduler first calibrates the reference traces to account for differences in Skills and execution configurations, then averages the calibrated traces to estimate the candidate batch's code-under-test submission intensity.

Here, k denotes a category of code under test with similar execution costs, and u is a discrete time offset relative to admission. Let $g_{b,k,p}(u)$ denote the capacity contribution per unit of submission intensity up to time u. We compute

$$\widehat{d}_{b,p}(u) = \sum_k \widehat{\lambda}_{b,k}(u)\, g_{b,k,p}(u), \qquad \overline{d}_{b,p}(u) = \widehat{d}_{b,p}(u) + \varepsilon_{b,p}(u). \quad (3)$$

The estimate $\widehat{d}$ is expressed in units of Worker capacity, rather than submission counts or observed utilization. The term ε is a conservative margin determined from historical residuals and errors arising from calibration across configurations.

3.4 Selecting the Next Batch under Capacity Constraints

Figure 3(d) illustrates the final selection step. Let $\overline{D}_{p,t}(u)$ denote the projected resource demand at a future time from currently running and queued tasks, together with batches already admitted in the current round. Let $C_p(t+u)$ denote the capacity of resource pool p. On the prediction grid $\mathcal{U} = \{0, \Delta, \ldots, H\}$, which is fixed for the scheduling round, admission of candidate batch b must satisfy

$$\overline{D}_{p,t}(u) + \overline{d}_{b,p}(u) \leq (1-\mu) C_p(t+u), \quad \forall p,\ u \in \mathcal{U}, \quad (4)$$

where $\mu \in [0,1)$ is the capacity margin. The algorithm examines candidate batches in a fixed priority order and updates the projected aggregate demand after each admission. When observations or historical evidence are insufficient, it falls back to conservative behavior. Algorithm 1 combines the four stages in Figure 3 into a single scheduling round.

Algorithm 1 Semantics-Aware Task Scheduling

Input: candidate batches $\mathcal{B}_t^{\text{wait}}$, available history $\mathcal{D}_t$, and current snapshot o_t.
Output: admission proposals to be validated by the executor.

```
Extract each batch's χ_b from o_t, estimate D̄, and initialize A ← ∅.
for each candidate batch b in fixed priority order do
    Construct V_t(b) from D_t; set ready ← false.
    for each tier V_t^(1)(b), V_t^(2)(b), V_t^(3)(b) in order do
        Select up to K references, check configuration differences, calibrate GPU usage profiles, and estimate d̄_b.
        if evidence is sufficient and the prediction is usable then set ready ← true and exit the inner loop.
    end for
    if ¬ready or baseline demand is unknown then apply a conservative fallback.
    else if Eq. (4) holds then set A ← A ∪ {b} and update the projected aggregate demand.
    else defer b.
    end for
Pass the proposals to the executor for revalidation against the execution-time state.
Write newly obtained execution records and prediction residuals to D for use in subsequent rounds.
```

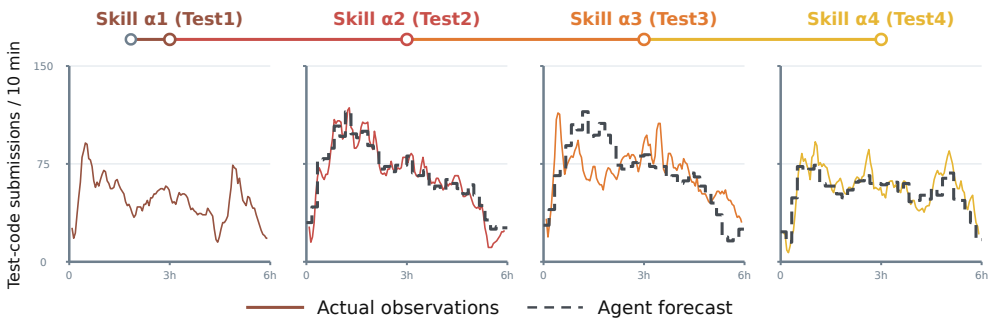


Figure 4: Temporal distributions of code-under-test submissions across Skill versions. Colored solid lines show observations, and black dashed lines show predictions.

4 Experiments

The first three experiments evaluate demand prediction and scheduling at three levels. The first varies only the Skill to assess whether the Scheduling Agent can accurately predict Worker usage profiles. The second examines whether these predictions remain accurate when the Skill, model configuration, and task composition change simultaneously. The third enables and disables the Scheduling Agent in production to evaluate its effect on Worker utilization and its ability to detect and respond appropriately to unexpected model-API events. We then report the composition of stored files and the results of residual-file cleanup.

4.1 Predicting Worker Usage Profiles across Skill Versions

The first experiment varies only the Skill. Given the differences between consecutive Skill versions and the execution history of the preceding batch, the Scheduling Agent predicts the temporal distribution of code-under-test submissions for the next batch. We compare these predictions with the observed submission traces to assess how accurately the agent predicts Worker usage profiles.

To quantify the difference between predicted and observed distributions, we use the first Wasserstein distance, also known as Earth Mover's Distance. We normalize the observed count y_i and predicted count $\widehat{y}_i$ in the i-th ten-minute time bin into probability masses $p_i = y_i / \sum_j y_j$ and $q_i = \widehat{y}_i / \sum_j \widehat{y}_j$, respectively, and define

$$D_W(p,q) = \Delta t \sum_{i=1}^{K} \left| \sum_{j=1}^{i} (p_j - q_j) \right|, \qquad \Delta t = 10 \text{ min},$$
$$S_W(p,q) = S_W(q,p) = 1 - \frac{D_W(p,q)}{H}, \quad H = K\Delta t = 6 \text{ h}. \tag{5}$$

Here, K is the number of time bins. D_W measures the minimum average temporal displacement required to transport the predicted probability mass to the observed distribution and is invariant to exchanging the two distributions. S_W converts this distance into a symmetric similarity score in $[0, 1]$, with higher values indicating closer temporal distributions.

Figure 4 presents the submission profiles for four consecutive Skill versions. The available predictions generally track the observations closely, indicating that the Scheduling Agent captures the temporal shape of the submission workload with reasonable accuracy. Submission profiles also exhibit similar patterns of evolution across consecutive Skill versions on the same Git branch. For the final three panels, which include predictions, the S_W scores are 0.990, 0.959, and 0.989, respectively, with a mean of 0.979.

4.2 Predicting Worker Usage Profiles under Multiple Configuration Changes

The second experiment varies the Skill, model configuration, and task composition simultaneously. Across consecutive batches with these changes, the Scheduling Agent uses the available historical information to predict the temporal distribution of code-under-test submissions for the next batch. We compare the predictions against the observed traces to assess whether prediction accuracy is maintained under simultaneous changes in multiple factors.

Figure 5 shows generally close agreement between the available predictions and observations across four consecutive batches, indicating that the Scheduling Agent can still predict submission distributions with reasonable accuracy when multiple factors change. For the final three panels, which include predictions, the S_W scores computed using Equation (5) are 0.944, 0.951, and 0.985, respectively, with a mean of 0.960. Higher scores indicate closer agreement between predicted and observed distributions. Together with Figure 4, these results suggest that consecutive Skill versions on the same Git branch often retain similar temporal distribution shapes.

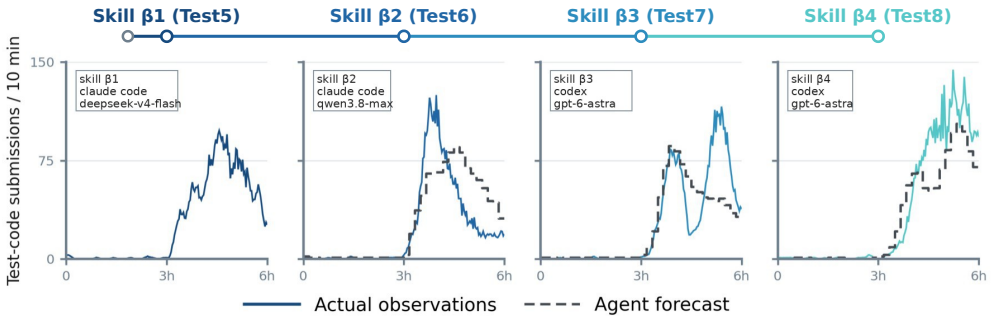


Figure 5: Temporal distributions of code-under-test submissions across four consecutive batches with simultaneous changes in multiple factors. Colored solid lines show observations, and black dashed lines show predictions.

To compare the temporal distributions across both experiments, Table 1 denotes the four observed distributions in Figure 4 by α_1–α_4 and those in Figure 5 by β_1–β_4. Primed columns denote the corresponding predicted distributions. We compute pairwise S_W scores among all eight observed distributions. Each predicted distribution is compared only with its corresponding observation; other pairings involving predictions are not evaluated.

Table 1: Wasserstein similarity (S_W) between the temporal submission distributions in Figures 4 and 5. Pairwise symmetric similarities are computed among the eight observed distributions; each off-diagonal pair is computed once and mirrored across the matrix diagonal. Primed columns report the similarity between each predicted distribution and its corresponding observed distribution. Gray dashes indicate prediction pairings for which no comparison is defined. Darker shades of yellow indicate higher similarity.

	Skill-only distributions (4.1)							Multi-factor distributions (4.2)						
	Observed				Forecast			Observed				Forecast		
row	α_1	α_2	α_3	α_4	α'_2	α'_3	α'_4	β_1	β_2	β_3	β_4	β'_2	β'_3	β'_4
α_1	1.000	0.958	0.973	0.962	–	–	–	0.684	0.751	0.660	0.628	–	–	–
α_2	0.958	1.000	0.955	0.935	0.990	–	–	0.657	0.724	0.633	0.601	–	–	–
α_3	0.973	0.955	1.000	0.975	–	0.959	–	0.697	0.764	0.673	0.641	–	–	–
α_4	0.962	0.935	0.975	1.000	–	–	0.989	0.722	0.789	0.698	0.666	–	–	–
β_1	0.684	0.657	0.697	0.722	–	–	–	1.000	0.931	0.970	0.944	–	–	–
β_2	0.751	0.724	0.764	0.789	–	–	–	0.931	1.000	0.909	0.876	0.944	–	–
β_3	0.660	0.633	0.673	0.698	–	–	–	0.970	0.909	1.000	0.965	–	0.951	–
β_4	0.628	0.601	0.641	0.666	–	–	–	0.944	0.876	0.965	1.000	–	–	0.985

4.3 Comparing Worker Utilization under FIFO and Semantics-Aware Scheduling

Scheduling baseline. The system's original deterministic scheduling policy is first-in, first-out (FIFO). Tasks are processed in the order in which they enter the waiting queue. If the task at the head of the queue cannot currently satisfy its hardware, execution-environment, API concurrency, or quota requirements, the scheduler waits for that task rather than reordering subsequent tasks based on their semantics or anticipated workload supply. Scheduling Agent OFF denotes FIFO alone. Scheduling Agent ON augments FIFO with semantics-aware scheduling that combines Skill-aware concurrency adjustment and admission control.

Resource utilization metric. For an observation at time t, let $B(t) = \sum_p B_p(t)$ and $I(t) = \sum_p I_p(t)$ denote the aggregate busy and idle capacity across all hardware resource pools, respectively. We define Worker utilization as

$$u(t) = \frac{B(t)}{B(t) + I(t)}. \tag{6}$$

Capacity is measured in device-seconds, so this metric captures aggregate Worker utilization across resource pools. Utilization of on-chip compute units is measured separately. For the long-term analysis, we partition the observation period into Scheduling Agent ON (FIFO + Scheduling Agent) and Scheduling Agent OFF (FIFO) intervals and retain intervals with sufficient task supply. For each retained set of valid intervals $\mathcal{I}_s$, where $s \in \{\text{on}, \text{off}\}$, we report the utilization range using the time-weighted 10th and 90th percentiles:

$$[P_{10,s}, P_{90,s}] = \text{Quantile}^{\Delta t}_{0.10,0.90}\left(\{u(t) : t \in \mathcal{I}_s\}\right),$$
$$\overline{u}_s = \frac{\sum_{t \in \mathcal{I}_s} u(t)\,\Delta t}{\sum_{t \in \mathcal{I}_s} \Delta t}. \tag{7}$$

Here, Δt is the duration of the observation window.

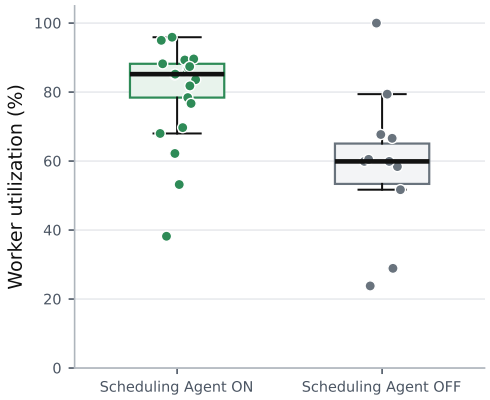


Figure 6: Worker utilization with the Scheduling Agent enabled and disabled. Scheduling Agent ON denotes FIFO + Scheduling Agent, whereas Scheduling Agent OFF denotes FIFO alone. Each point represents the mean Worker utilization for one Scheduling Agent state within an observation day. Boxes indicate the interquartile range, and black lines indicate the median.

To summarize the daily execution traces in a comparable aggregate view, we group observations into FIFO and FIFO + Scheduling Agent according to whether the Scheduling Agent is enabled. We first compute the mean Worker utilization for each state within each observation day, then use these daily per-state means as the observations in the box-and-scatter plot in Figure 6.

The figure includes daily means from 19 observation days for FIFO + Scheduling Agent and 11 observation days for FIFO. With all observations retained, the medians of these daily means are 85.0% and 59.9%, respectively. After excluding the two lowest observations in the FIFO + Scheduling Agent group and the highest observation in the FIFO group, 17 and 10 observations remain, with medians of 85.2% and 59.9%, respectively.

The two low observations excluded from the FIFO + Scheduling Agent summary statistics correspond to Day 17 and Day 18. External model services returned errors including "at capacity," 429, 500, and 503, prompting the Scheduling Agent to reduce concurrency for the affected APIs. The lower utilization therefore reflects a protective reduction in load during service disruptions. The highest observation excluded from the FIFO summary statistics corresponds to Day 6. After the Scheduling Agent was disabled, FIFO admitted too many operator-generation tasks, keeping Workers saturated for an extended period and causing code-under-test submissions to accumulate. All three exceptional observations remain visible in the figure so that they can be cross-referenced with the daily descriptions. Semantics-aware admission dynamically adjusts utilization in response to external service conditions and task supply to maintain end-to-end operation.

4.4 Residual-File Cleanup

During long-running operation, agents occasionally write artifacts to incorrect paths that are not covered by cleanup routines based on fixed rules. Task types also evolve continuously; unless cleanup rules are updated promptly, some newly generated artifacts may remain uncleared for extended periods. The Scheduling Agent uses file semantics to clean up these residual artifacts. It removed approximately 2.2 TB of residual files from incorrect paths and freed about 380,000 inodes.

5 Discussion and Limitations

Applicability and cost trade-offs. The Scheduling Agent is useful when scheduling decisions require interpreting semantic information available within the system, as in the Agent–Worker operator-generation setting studied here. Although it introduces additional inference overhead, it substantially improves Worker utilization at the deployment scale considered in this work. We focus on supply-side scheduling, where agents continuously generate work that Workers execute asynchronously. Existing approaches do not provide a common baseline directly suited to this setting. We therefore use FIFO as a straightforward baseline to evaluate the benefit of semantic information and limit our comparison to this baseline.

Why not control Worker utilization by pausing agents? We do not stabilize Worker utilization by repeatedly pausing and resuming agents that are already performing operator-generation tasks. In our setup, subagent sessions cannot be restored after such pauses, substantially reducing comparability between executions with and without pausing. The system therefore adjusts only the admission of new tasks and subsequent concurrency, without interrupting running tasks in ways that would lose execution state.

Future directions. This mechanism could be extended to Agent-as-a-Service (AaaS), where a service hosts agent tasks from multiple users. In this setting, the scheduler must consider not only individual tasks but also resource isolation, quotas, and failure recovery across users. Similar approaches could support other generation workflows, such as Verilog generation and data processing. These workflows may consume GPU, CPU, memory, and storage resources simultaneously, with different tasks exhibiting different resource demands and bottlenecks.

6 Conclusion

We present a semantics-aware task scheduling approach that uses semantic information from Skills, execution configurations, and historical traces to predict code-under-test submission distributions and Worker demand, thereby controlling task concurrency and admission. Our observations show that Skill versions on the same branch often exhibit similar Worker usage patterns, which the Scheduling Agent can exploit to predict submission distributions and Worker demand with reasonable accuracy. Across 28 observation days covering 29,426 operator-generation tasks and 1,380,794 code-under-test submissions, the median Worker utilization was 85.2% for FIFO + Scheduling Agent and 59.9% for FIFO. The Scheduling Agent also uses semantic information to identify residual files for cleanup, helping reclaim disk space and inodes.

Data Scope and AI Assistance

Our data include production logs collected over more than a month and other records gathered during six months of system operation. These include configurations, execution traces, and generated artifacts, totaling tens of terabytes. We used AI assistance for language editing and figure refinement. The author reviewed all content.

Ethics and Reproducibility

All reported logs and statistics come from real system operation. Model services change frequently as the field evolves, making it difficult to recreate identical runtime environments. Exact repro-

duction of every result therefore cannot be guaranteed. This work introduces no additional ethical risks.

REFERENCES

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2207.10397.

J. G. Dai, Tianze Deng, Yueying Li, and Tianyi Peng. Throughput-optimal scheduling algorithms for LLM inference and AI agents. *arXiv preprint arXiv:2504.07347*, 2025. URL https://arxiv.org/abs/2504.07347.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA Agent: Large-scale agentic RL for high-performance CUDA kernel generation. *arXiv preprint arXiv:2602.24286*, 2026. URL https://arxiv.org/abs/2602.24286.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 16344–16376, 2022. URL https://arxiv.org/abs/2205.14135.

Dongxin Guo, Jikun Wu, and Siu Ming Yiu. SAGA: Workflow-atomic scheduling for AI agent inference on GPU clusters. *arXiv preprint arXiv:2605.00528*, 2026. URL https://arxiv.org/abs/2605.00528.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.06770. Oral.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023. URL https://arxiv.org/abs/2309.06180.

Gang Liao, Hongsen Qin, Ying Wang, et al. KernelEvolve: Scaling agentic kernel coding for heterogeneous AI accelerators at Meta. *arXiv preprint arXiv:2512.23236*, 2025. URL https://arxiv.org/abs/2512.23236.

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)*, 2024. URL https://arxiv.org/abs/2405.19888.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, Dheeraj Peri, Ouye Xie, Aditya Kane, Moshe Maor, Michael Behar, Triston Cao, Rishabh Mehta, Vartika Singh, Vikram Sharma Mailthody, Terry Chen, Zihao Ye, Hanfeng Chen, Tianqi Chen, Vinod Grover, Wei Chen, Wei Liu, Eric Chung, Luis Ceze, Roger Bringmann, Cyril Zeller, Michael Lightstone, Christos Kozyrakis, and Humphrey Shi. SOL-ExecBench: Speed-of-light benchmarking for real-world GPU kernels against hardware limits. *arXiv preprint arXiv:2603.19173*, 2026. URL https://arxiv.org/abs/2603.19173.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2308.03688.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Autellix: An efficient serving engine for LLM agents as general programs. *arXiv preprint arXiv:2502.13965*, 2025. URL https://arxiv.org/abs/2502.13965.

Kai Mei, Xi Zhu, Wujiang Xu, Wenyue Hua, Mingyu Jin, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. AIOS: LLM agent operating system. In *Proceedings of the Conference on Language Modeling (COLM)*, 2025. URL https://arxiv.org/abs/2403.16971. Full paper.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026. URL https://arxiv.org/abs/2601.11868.

Alexander Novikov et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025. URL https://arxiv.org/abs/2506.13131.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025. URL https://arxiv.org/abs/2502.10517.

Tara Saba, Zhiyang Chen, Jikai Jason Li, Anne Ouyang, Xujie Si, and Fan Long. CuTeGen: An LLM-based agentic framework for generation and optimization of high-performance GPU kernels using CuTe. *arXiv preprint arXiv:2604.01489*, 2026. URL https://arxiv.org/abs/2604.01489.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 68539–68551, 2023. URL https://arxiv.org/abs/2302.04761.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2407.16741.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *Proceedings of the First Conference on Language Modeling (COLM)*, 2024. URL https://arxiv.org/abs/2308.08155.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 50528–50652, 2024. doi: 10.52202/079017-1601. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2210.03629.

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. *arXiv preprint arXiv:2504.20068*, 2025. URL https://arxiv.org/abs/2504.20068. Earlier versions circulated under the title Tempo.

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 *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 62557–62583, 2024. doi: 10.52202/079017-2000. URL https://arxiv.org/abs/2312.07104.

Yusheng Zheng, Yanpeng Hu, Wei Zhang, and Andi Quinn. Towards agentic OS: An LLM agent framework for Linux schedulers. *arXiv preprint arXiv:2509.01245*, 2025. URL https://arxiv.org/abs/2509.01245. Journal reference: MLforSystem 2025.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2307.13854.

A Scheduling Problem and Control Objective

A.1 Control State and Read-Only Queries

A nonstationary, partially observed scheduling environment. Let the complete system state be

$$S_t = (\mathcal{B}_t, \mathcal{Q}_t, \mathcal{R}_t, \kappa_t, \xi_t), \qquad S_{t+1} \sim \mathsf{P}_t(\cdot \mid S_t, a_t, \ell_t). \tag{8}$$

$\mathcal{B}_t$ contains configurations and states of batches and their operator-generation tasks; $\mathcal{Q}_t$ contains waiting and running code-under-test; $\mathcal{R}_t$ contains Worker capacity and occupancy; and κ_t is the active control configuration. The latent variable ξ_t includes development progress, remaining code-execution time, and internal model-service state. The unknown transition law P_t may change with workload composition and service quality. ℓ_t represents query, inference, and execution latency, making explicit the delay between observation and action effect.

For resource pool p, the number of unfinished code-under-test items satisfies

$$q_{p,t+1} = q_{p,t} + N_{p,t} - D_{p,t}. \tag{9}$$

Here $q_{p,t}$ includes waiting and running code-under-test, and $D_{p,t}$ counts items leaving this set after completion, failure, or cancellation; starting execution alone does not count toward $D_{p,t}$. Control affects $N_{p,t}$ mainly by changing the set of operator-generation tasks launched subsequently. Already-started tasks may continue submitting code, so restricting new admission does not immediately reduce supply to zero or remove an existing backlog.

Snapshot-based adaptive queries. The observation in each round is $o_t = (x_t, z_t)$: x_t contains structured information such as code-under-test queue length and waiting time, Worker occupancy, active operator-generation tasks, account usage, and current configuration; z_t contains Skills, execution logs, and error text read through tools. The same observation can correspond to different latent states: an idle GPU may indicate that several operator-generation tasks are in an analysis phase or that a model service failure is blocking progress. Idle capacity alone is therefore insufficient to predict the effect of increasing concurrency.

The Agent retrieves needed records from recent history h_t and issues read-only queries incrementally:

$$h_t^{(0)} = \mathrm{Retrieve}(h_t), \qquad h_t^{(k+1)} = h_t^{(k)} \oplus (v_{t,k}, y_{t,k}, \tau_{t,k}). \tag{10}$$

$v_{t,k}$ is a query, $y_{t,k}$ its returned content, and $\tau_{t,k}$ its timestamp. For example, when code supply is low, the Agent first checks model-service health and then inspects the relevant Skill and recent task records; when a queue grows continuously, it checks backlog composition, recent admissions, and prior actions.

Complete control objective. Given a common operator-generation task list, resource budget, and deadline T, the objective is to complete more admissible operator-generation tasks while obtaining timely feedback. Writing the control policy as π, the runtime objective is

$$\max_{\pi} \ \mathbb{E}_\pi[\alpha C_T - \beta W_T + \gamma U_T - \delta R_T - \eta K_T], \qquad W_T = \int_0^T q^{\mathrm{wait}}(\tau)\,\mathrm{d}\tau. \tag{11}$$

C_T is the number of distinct operator-generation tasks passing fixed acceptance checks before the deadline; q^{wait} is the number of waiting code-under-test items; U_T is capacity-weighted long-run GPU occupancy; R_T is the restart rate for error-triggered operator-generation tasks; and K_T is scheduling cost. Unstarted code-under-test contributes to W_T until the window closes. The completion rate is C_T/M, where M is the full set of operator-generation tasks allocated in advance and arriving on schedule for the experiment block.

Operator-generation tasks and code supply. Let t denote a control round and $\mathcal{J}_t^+$ the set of operator-generation tasks that ran during its interval. If task j produces $N_{j,t}$ code-under-test items in the interval, then

$$N_t = \sum_{j \in \mathcal{J}_t^+} N_{j,t}. \tag{12}$$

$N_{j,t}$ is unknown at the start of the round and changes with Skill stage, model response, Harness behavior, and test feedback. The same number of active operator-generation tasks can therefore produce different submission rates, and the same number of code submissions can represent different GPU workloads because runtime and resource usage vary.

The controller reads queues, resource occupancy, API events, and operator-generation context, and adjusts the concurrency allowed for model-service accounts and new-task admission. The underlying dispatcher still assigns submitted code-under-test to Workers. GPU utilization in this paper means capacity-weighted Worker occupancy; Workers occupied during compilation preparation are included, while chip-level compute utilization is reported separately.

B Long-Run Operation and Resource Timeline

Figure 7 summarizes Worker utilization and queued-task counts during long-run operation; the daily pages that follow show the changes over time. The 28 retained observation dates are numbered Day 1–28 in chronological observation order, with calendar dates shown in the daily panels. The record contains 29,426 operator-generation tasks and 1,380,794 code submissions.

The operator-generation tasks use four Harnesses—Codex, Claude Code, DSH, and KimiCode—and models including GPT-5.6-Sol, GPT-6-Astra, Qwen3.8-Flash/Max, DeepSeek-Flash, DeepSeek-V4-Flash, Kimi K3, Doubao-Seed-Evolving, and GLM-5.2/5.3. Model services include OpenAI, DeepSeek, Alibaba Cloud, Kimi, and VolcEngine.

B.1 Scheduling States and Resource Timeline

Figure 7 aggregates Worker utilization and end-of-window queued tasks in 30-minute windows. Worker utilization is the fraction of execution-pool capacity in use; missing or invalid records are blank. Daily notes record task types, models, Harnesses, Skills, service conditions, and Scheduling Agent toggles and restarts, and use them to explain queue and utilization changes. Following Section 4.3, Scheduling Agent OFF denotes FIFO alone, while Scheduling Agent ON adds semantic scheduling to FIFO; both states retain FIFO queue order and API-capacity constraints. The tables and daily panels below use these labels consistently. Days 1–22 primarily show concurrency control, service restarts, and model-API rate limiting; Days 23–28 provide comparison observations with Scheduling Agent OFF (FIFO).

The daily system-operation statistics above are arranged as 14 consecutive two-day figures. Each caption identifies its Day range, and the corresponding notes are retained below each figure.

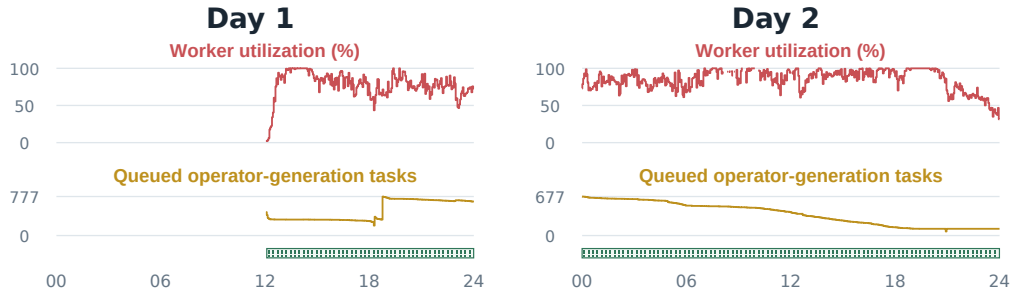


Figure 8: System operation record for Day 1 and Day 2.

Day 1: Operators were mainly FA/Attention, MHC, GEMM/Matmul, and TopK/Sampling. The model was deepseek-v4-flash with Claude Code and the official DeepSeek service. The Skill added advanced-operator pipeline knowledge, with six-hour task limits. After submissions began around noon, Scheduling Agent admitted an initial group and held back the remainder; it resumed admissions as the peak fell below 90%. With sufficient tasks, Worker utilization was 61.6%–99.4% (time-weighted mean 78.4%).

Day 2: The system primarily continued executing the tests submitted on Day 1. Long unattended periods still maintained high Worker utilization with Scheduling Agent enabled. With sufficient tasks, utilization was 64.4%–99.9% (mean 85.0%).

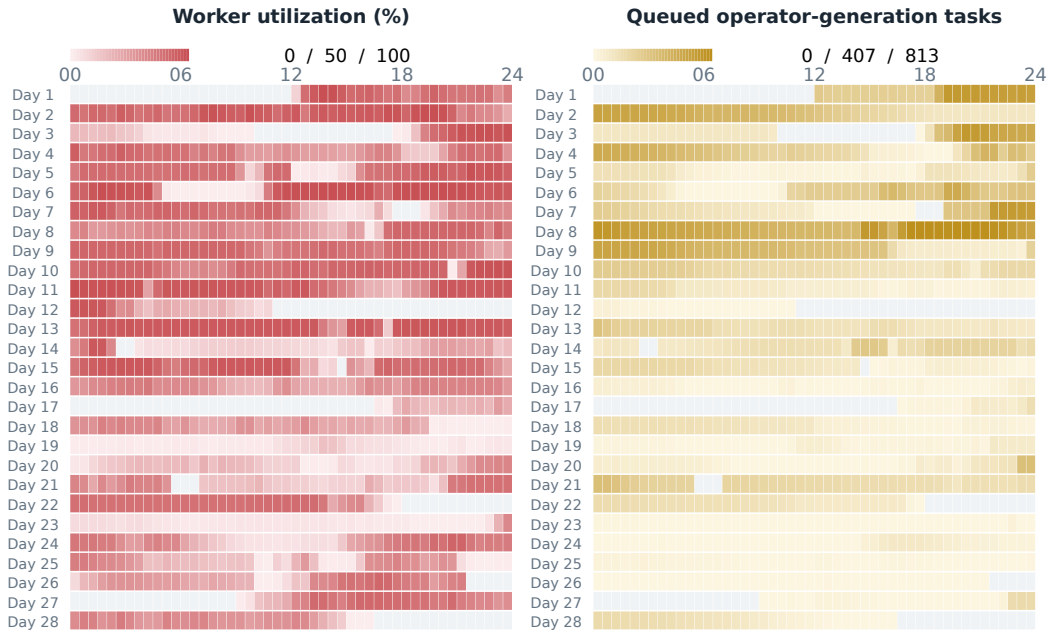


Figure 7: Overview of Worker utilization and queued tasks during long-run operation. The left panel shows Worker utilization and the right panel shows queued operator-generation tasks; each row corresponds to one observation day (Day 1–28).

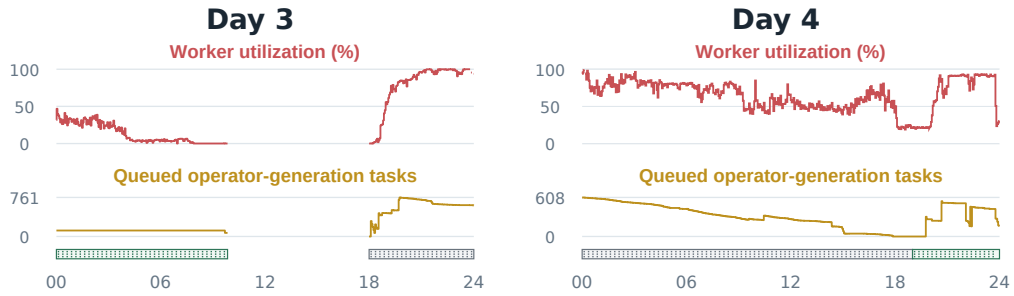


Figure 9: System operation record for Day 3 and Day 4.

Day 3: Operators were mainly FA/Attention, MHC, Norm, and IO/Quant. The deepseek-v4-flash/Claude Code configuration added hardware-information retrieval to the Skill. Scheduling Agent was active during the daytime data segment, but task supply was limited; it was disabled after about 18:00. During the OFF period, utilization was 24.3%–100.0% (mean 79.4%).

Day 4: Remaining FA/Attention, MHC, Norm, and IO/Quant tasks from Day 3 continued to run. Scheduling Agent was re-enabled around 19:00 before a new batch arrived. With sufficient tasks, ON utilization was 28.3%–92.9% (mean 76.7%); OFF utilization was 46.1%–84.5% (mean 66.6%).

Table 2: Worker utilization on Days 1–28 under FIFO + Scheduling Agent (Scheduling Agent ON) and FIFO (Scheduling Agent OFF).

	FIFO + Scheduling Agent (Scheduling Agent ON)		FIFO (Scheduling Agent OFF)	
Day	Range (%)	Time-weighted mean (%)	Range (%)	Time-weighted mean (%)
Day 1	61.6–99.4	78.4	–	–
Day 2	64.4–99.9	85.0	–	–
Day 3	–	–	24.3–100.0	79.4
Day 4	28.3–92.9	76.7	46.1–84.5	66.6
Day 5	84.3–97.8	89.3	–	–
Day 6	83.1–99.9	95.0	Near 100% for an extended period*	
Day 7	74.6–92.3	83.6	18.8–69.3	51.7
Day 8	61.4–95.8	86.1	44.2–70.7	58.4
Day 9	71.4–95.8	87.0	–	–
Day 10	71.5–97.3	88.2	–	–
Day 11	49.7–98.1	85.2	–	–
Day 12	94.6–99.8	95.9	–	–
Day 13	78.6–95.6	89.6	–	–
Day 14	24.7–94.5	69.7	9.7–49.7	28.9
Day 15	77.8–95.9	87.4	–	–
Day 16	47.9–73.0	62.2	–	–
Day 17	27.7–54.5	38.2	–	–
Day 18	39.3–69.7	53.2	–	–
Day 19	–	–	–	–
Day 20	–	–	–	–
Day 21	39.1–88.0	68.0	14.2–33.0	23.8
Day 22	54.7–90.2	81.8	–	–
Day 23	–	–	–	–
Day 24	–	–	42.5–87.1	67.7
Day 25	–	–	58.9–62.0	59.9
Day 26	–	–	–	–
Day 27	–	–	53.6–68.7	60.5
Day 28	–	–	49.2–70.2	59.9

Note: All values are transcribed from the Day 1–28 notes below, using the ranges and time-weighting reported for each day. A dash means that no range or mean is reported for the corresponding state, including days without that state or with insufficient task supply; it does not mean zero. *The two Day 6 Scheduling Agent OFF columns combine the saturation description of utilization remaining near 100% for an extended period.

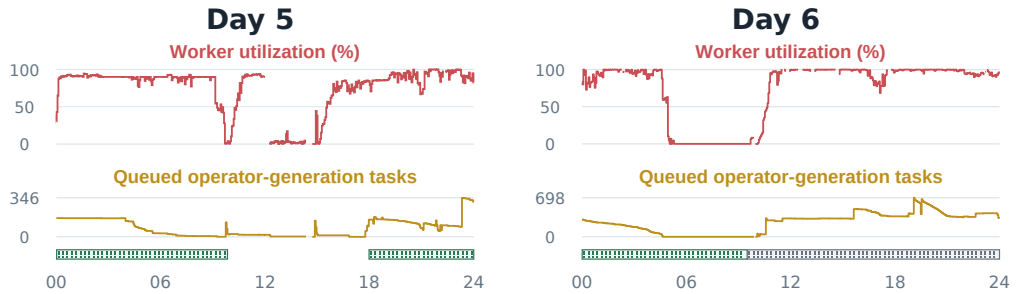


Figure 10: System operation record for Day 5 and Day 6.

Day 5: Operators were mainly Norm, GEMM/Matmul, TopK/Sampling, and IO/Quant. The deepseek-v4-flash/Claude Code Skill added libdevice optimization, with six- and ten-hour limits. Scheduling Agent deliberately reduced concurrency overnight to prevent backlog, then used historical utilization to choose a more aggressive parallelism level later. Frequent service updates and restarts around noon caused utilization spikes. ON utilization was 84.3%–97.8% (mean 89.3%).

Day 6: Operators were mainly RWKV/FLA, TopK/Sampling, Norm, and GEMM/Matmul. The deepseek-v4-flash/Claude Code setup used DeepSeek services and compatible hosted endpoints; the Skill added libdevice optimization and hardware-difference notes. After a service restart around 10:00, Scheduling Agent remained OFF and FIFO admitted tasks in arrival order. Worker utilization stayed near 100% for an extended period, producing a backlog. ON utilization was 83.1%–99.9% (mean 95.0%).

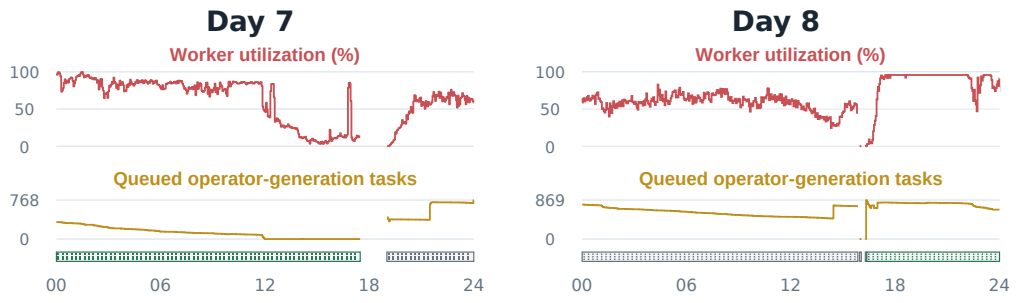


Figure 11: System operation record for Day 7 and Day 8.

Day 7: Operators were mainly TopK/Sampling, MHC, FA/Attention, and Norm. Models included deepseek-v4-flash and gpt-5.6-sol, with Claude Code and Codex. Scheduling Agent was ON before noon, then task supply diminished; after an 18:00 service restart it remained OFF. ON utilization was 74.6%–92.3% (mean 83.6%); OFF utilization was 18.8%–69.3% (mean 51.7%).

Day 8: Day 7's TopK/Sampling, MHC, FA/Attention, and Norm tasks continued. Scheduling Agent was OFF until service recovery, then restarted around 16:22; utilization rose as the queue was consumed. ON utilization was 61.4%–95.8% (mean 86.1%); OFF utilization was 44.2%–70.7% (mean 58.4%).

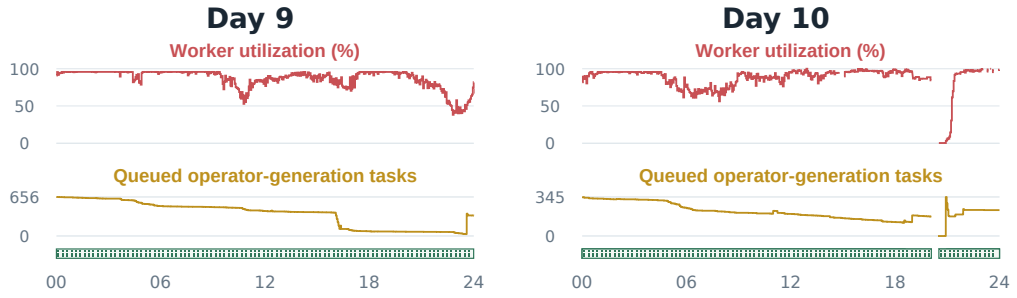


Figure 12: System operation record for Day 9 and Day 10.

Day 9: Operators were mainly FA/Attention, Norm, and GEMM/Matmul. Scheduling Agent stayed ON during valid data segments; utilization remained high while the queue gradually drained. ON utilization was 71.4%–95.8% (mean 87.0%).

Day 10: Operators were mainly RWKV/FLA, TopK/Sampling, FA/Attention, and Norm. Models included deepseek-v4-flash and gpt-5.6-sol with Claude Code and Codex. The Skill refined configuration tuning and hardware-difference knowledge. ON utilization was 71.5%–97.3% (mean 88.2%).

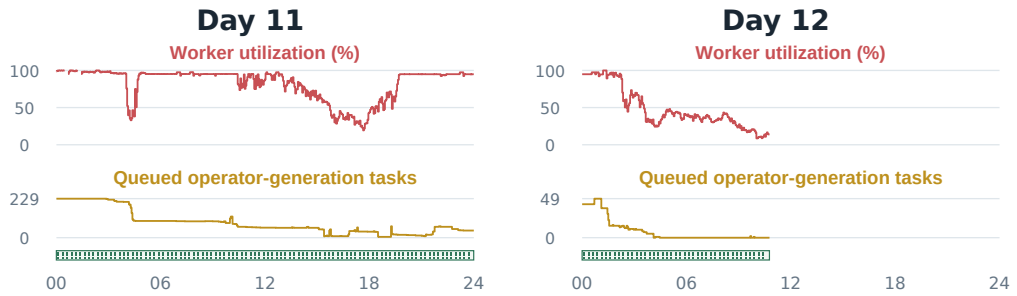


Figure 13: System operation record for Day 11 and Day 12.

Day 11: Operators were mainly FA/Attention, MHC, and RWKV/FLA. The Skill added anti-cheating checks, design versions, and search optimization. Agents initially produced extensive textual designs, so Scheduling Agent intentionally kept utilization moderate before submissions increased. ON utilization was 49.7%–98.1% (mean 85.2%).

Day 12: Operators were mainly FA/Attention, RWKV/FLA, GEMM/Matmul, and Norm. The Skill added database knowledge and workflow components. Tasks completed quickly before noon, and the small workload made lower utilization expected. ON utilization was 94.6%–99.8% (mean 95.9%).

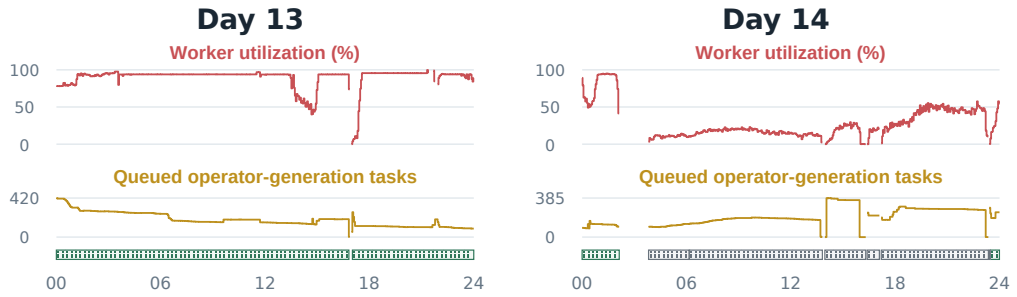


Figure 14: System operation record for Day 13 and Day 14.

Day 13: FA/Attention, TopK/Sampling, Norm, and IO/Quant tasks continued. Scheduling Agent stayed ON and the queue declined slowly; one Worker reached exactly 100% utilization, producing a nearly flat segment. ON utilization was 78.6%–95.6% (mean 89.6%).

Day 14: The prior batch continued, while an overnight service-cluster issue caused several restarts and utilization fluctuations. The OFF intervals used FIFO and had lower utilization; Scheduling Agent was re-enabled near midnight. ON utilization was 24.7%–94.5% (mean 69.7%); OFF utilization was 9.7%–49.7% (mean 28.9%).

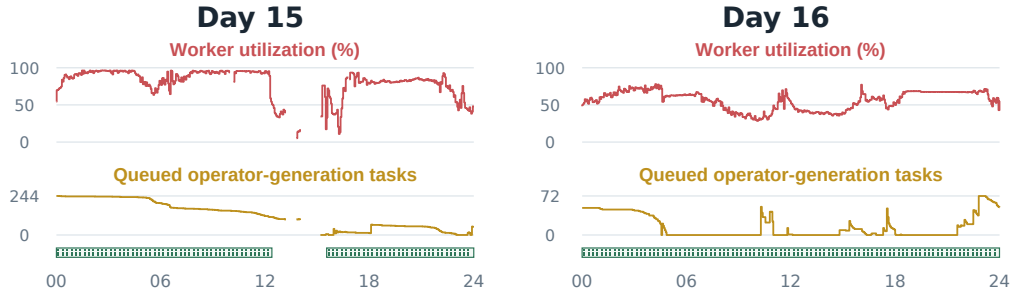


Figure 15: System operation record for Day 15 and Day 16.

Day 15: Operators were mainly FA/Attention, RWKV/FLA, IO/Quant, and TopK/Sampling. The Skill removed database integration, refined Triton optimization, and improved lower-bound estimation. Scheduling Agent stayed ON with high utilization: 77.8%–95.9% (mean 87.4%).

Day 16: Operators were mainly FA/Attention, Other, IO/Quant, and RWKV/FLA. Scheduling Agent stayed ON, but the workload was small and often left the queue empty. ON utilization was 47.9%–73.0% (mean 62.2%).

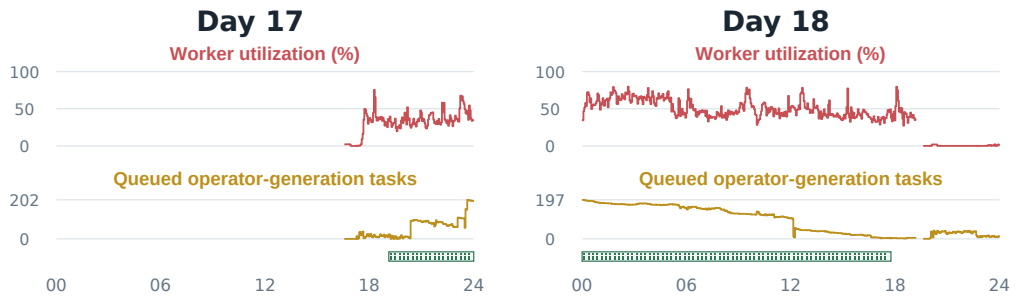


Figure 16: System operation record for Day 17 and Day 18.

Day 17: Operators were mainly FA/Attention, RWKV/FLA, IO/Quant, and GEMM/Matmul. Models included deepseek-v4-flash and gpt-6-astra. External service errors (at capacity, 429, 500, and 503) led Scheduling Agent to reduce gpt-6-astra concurrency; the queue nevertheless continued to drain. ON utilization was 27.7%–54.5% (mean 38.2%).

Day 18: Day 17 tasks continued and gradually completed. The same external-service constraints applied. ON utilization was 39.3%–69.7% (mean 53.2%).

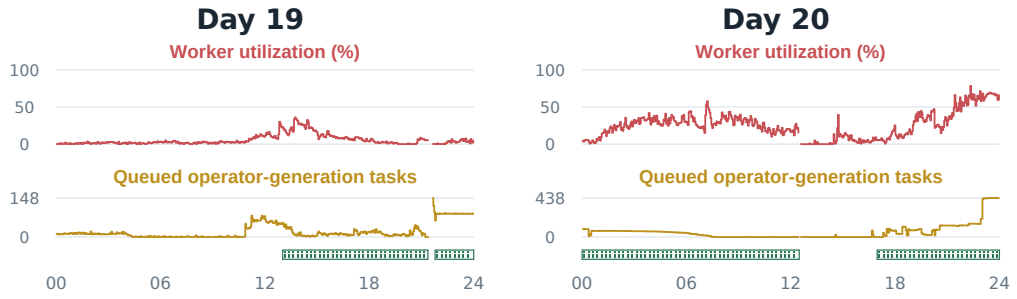


Figure 17: System operation record for Day 19 and Day 20.

Day 19: Operators were mainly FA/Attention, TopK/Sampling, Norm, and IO/Quant. Scheduling Agent remained ON, but the workload was small and some gpt-6-astra tasks had reduced parallelism because of API quality.

Day 20: Operators were mainly FA/Attention, TopK/Sampling, GEMM/Matmul, and Norm. Models included qwen3.8-flash, deepseek-v4-flash, doubao-seed-evolving, gpt-5.6-sol, and gpt-6-astra. Scheduling Agent behavior was similar to Day 19 because unstable OpenAI service quality limited parallelism. The workload was insufficient to expose a scheduling advantage.

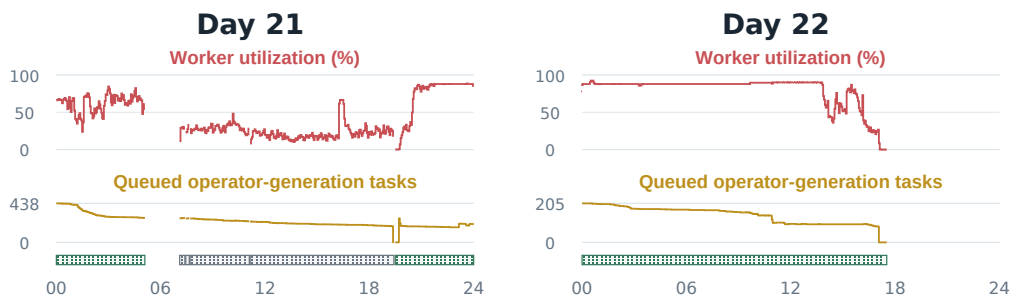


Figure 18: System operation record for Day 21 and Day 22.

Day 21: Operators were mainly FA/Attention, RWKV/FLA, GEMM/Matmul, and Norm. A service outage around 06:00 delayed Scheduling Agent startup; after a restart around 19:00, utilization remained high until the batch completed. ON utilization was 39.1%–88.0% (mean 68.0%); OFF utilization was 14.2%–33.0% (mean 23.8%).

Day 22: Operators were mainly FA/Attention, IO/Quant, Norm, and RWKV/FLA. A batch completed around 14:00, after which cancellations and service shutdown gradually reduced utilization. ON utilization was 54.7%–90.2% (mean 81.8%).

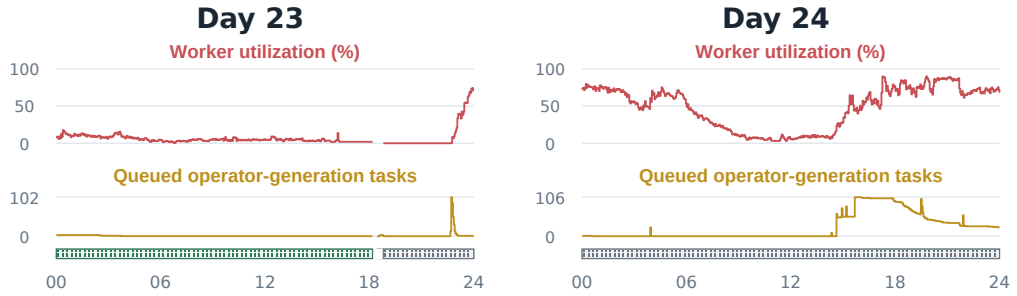


Figure 19: System operation record for Day 23 and Day 24.

Day 23: Operators were mainly FA/Attention, RWKV/FLA, and IO/Quant. Scheduling Agent was disabled from this day onward, retaining FIFO for comparison. Under FIFO, utilization was generally lower and variable; the workload was insufficient for a daily interval summary.

Day 24: Operators were mainly FA/Attention, TopK/Sampling, Norm, and IO/Quant. Only FIFO was used. OFF utilization was 42.5%–87.1% (mean 67.7%).

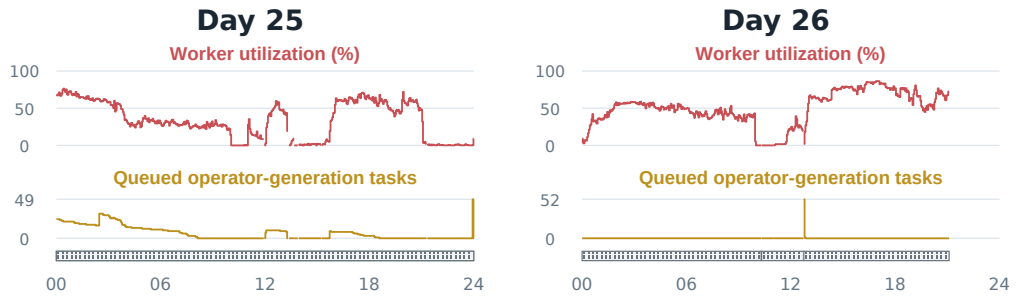


Figure 20: System operation record for Day 25 and Day 26.

Day 25: Operators were mainly FA/Attention, IO/Quant, RWKV/FLA, and Other. Only FIFO was used; removing the fixed-iteration constraint allowed agents to continue iterating based on quality and convergence. OFF utilization was 58.9%–62.0% (mean 59.9%).

Day 26: Operators were mainly FA/Attention, TopK/Sampling, Norm, and IO/Quant. Only FIFO was used, with low and variable utilization; task supply was insufficient for a daily interval summary.

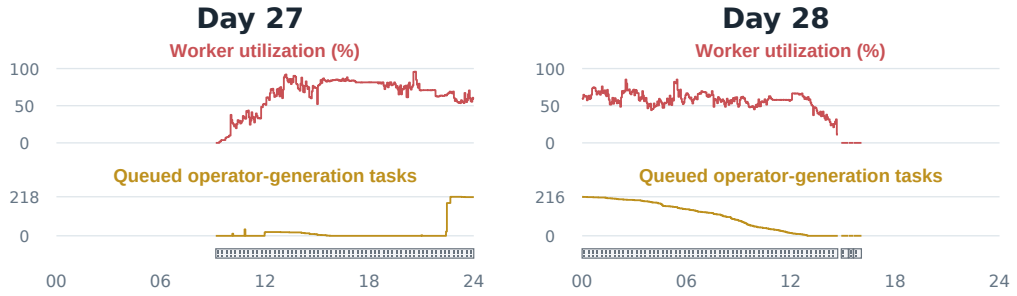


Figure 21: System operation record for Day 27 and Day 28.

Day 27: Operators were mainly FA/Attention, IO/Quant, and RWKV/FLA. Only FIFO was used; queue order and capacity constraints remained active without semantic concurrency adjustment. OFF utilization was 53.6%–68.7% (mean 60.5%).

Day 28: Day 27's FA/Attention, IO/Quant, and RWKV/FLA tasks continued across the date boundary. FIFO remained active and utilization declined as the remaining workload was consumed. OFF utilization was 49.2%–70.2% (mean 59.9%).

C File Types, Storage, and Inode Composition

We counted 3.242508 TB across several storage volumes during six months of operation, including 16,003 root objects; ordinary files account for 2.994737 TB of logical size. File purpose was classified from path semantics, format features, and restricted header checks.

The sample contains 75,752,519 unique inodes, including 59,027,054 regular files. Test inputs and tensors account for 42.997% of allocated space, profiling data for 21.054%, and compiled artifacts and work caches for 9.814%. Source and tests, documentation and prompts, and directories and links account for 25.991%, 22.351%, and 22.079% of inodes, respectively. Space and inode composition differ: these three categories occupy only 5.8%, 2.7%, and 2.1% of allocated space but consume many inodes.

D Code-Submission Distributions across Task Batches

Figure 23 summarizes the submission timing of the 47 retained task batches.

File type	Files	Size (GB)	Size (%)	Inodes	Inodes (%)
Test & tensor data	13,684	1,394.189	43.0%	13,684	0.018%
Profiler recordings	126,865	682.668	21.1%	126,865	0.167%
Build & working caches	6,226,840	318.235	9.8%	6,226,840	8.2%
Source code & tests	19,688,469	187.166	5.8%	19,688,469	26.0%
Version-control data	1,041,595	154.953	4.8%	1,041,595	1.4%
Trace	116,477	137.231	4.2%	116,477	0.154%
Gen. task & code records	11,207,436	128.507	4.0%	11,207,436	14.8%
Documentation & prompts	16,931,392	86.147	2.7%	16,931,392	22.4%
Directories & links	0	69.286	2.1%	16,725,465	22.1%
Execution logs	1,656,511	35.284	1.1%	1,656,511	2.2%
Runtime dependencies	1,872,294	32.755	1.0%	1,872,294	2.5%
Crash dumps	582	7.826	0.241%	582	<0.001%
Images & media	134,395	6.219	0.192%	134,395	0.177%
Packaged archives	10,249	2.041	0.063%	10,249	0.014%
Unresolved content	265	0.001	<0.001%	265	<0.001%
Total	**59,027,054**	**3,242.508**	**100.0%**	**75,752,519**	**100.0%**



Figure 22: Space and inode composition by file purpose in task directories. The sample contains 59,027,054 regular files, 75,752,519 unique inodes, and 16,003 root objects, with 3.242508 TB allocated. The table and pies use the same categories; inode counts include directories, links, and special objects, whereas Files counts regular files only.

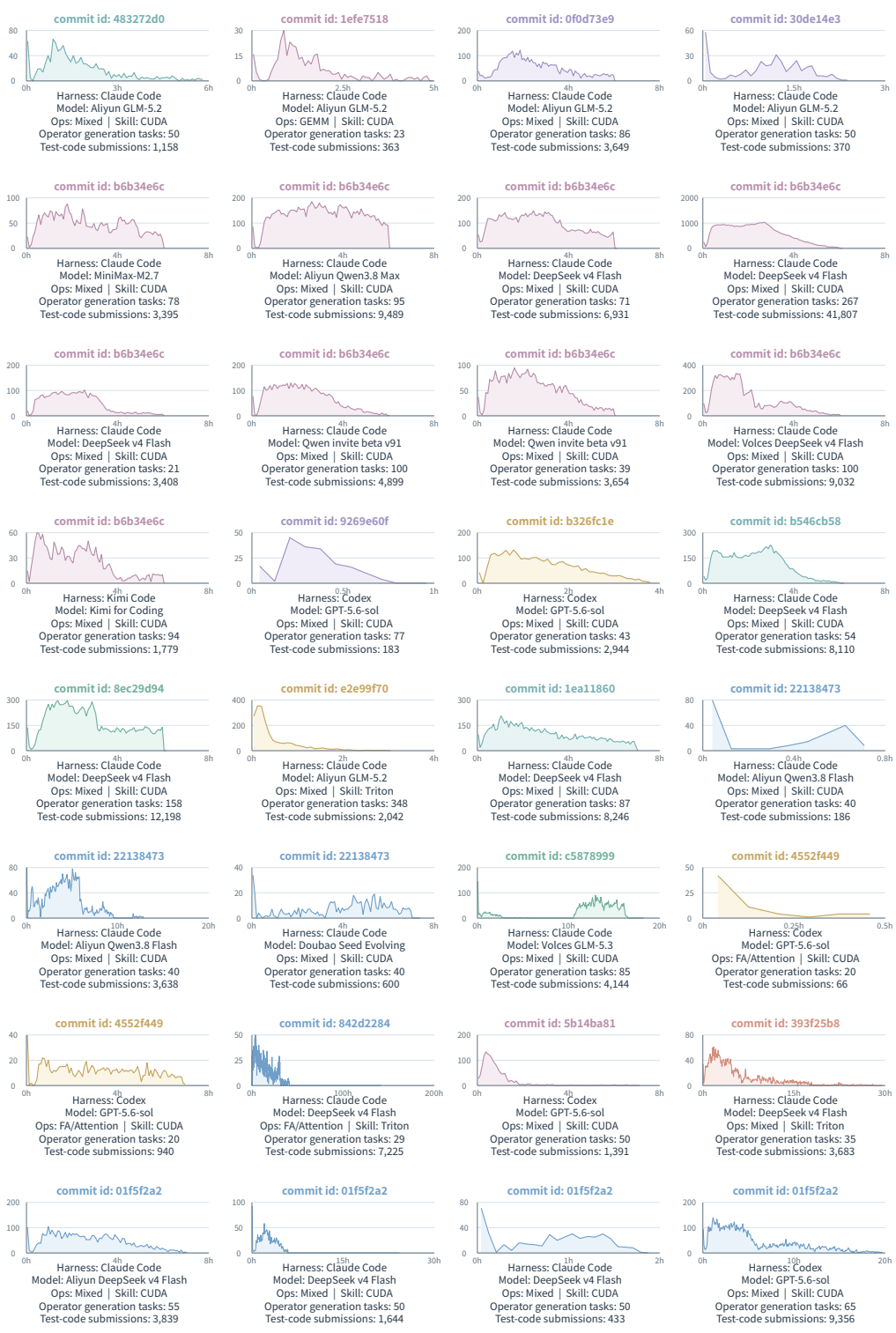


Figure 23: Code-submission distributions across task batches. Each page contains at most eight rows with four panels per row.

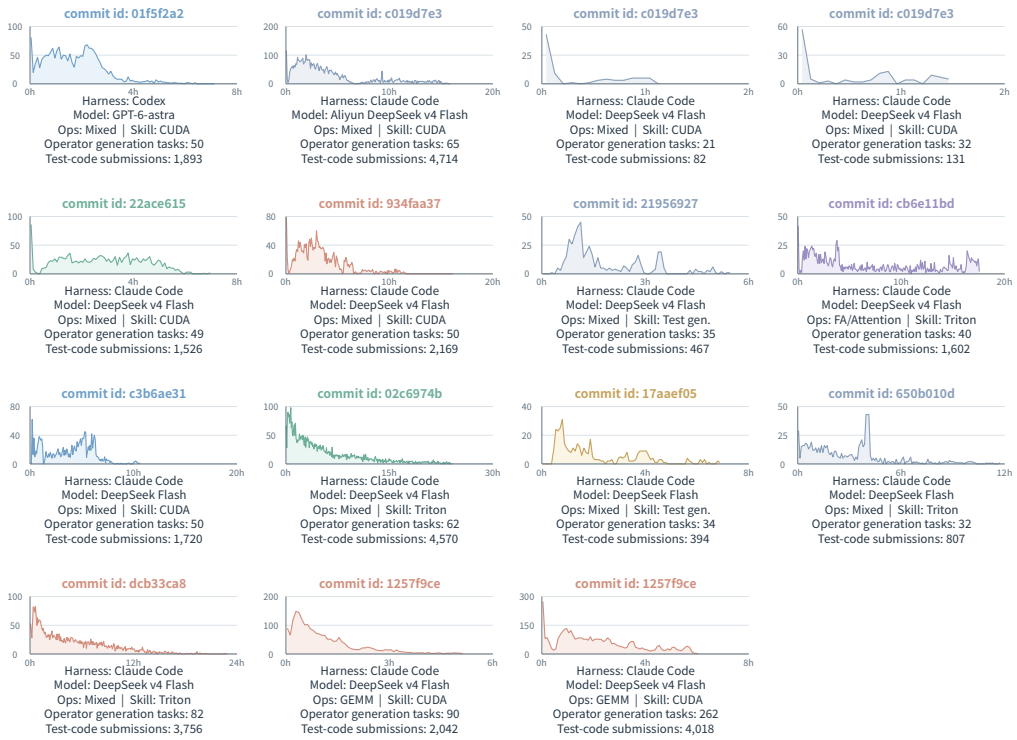


Figure 23: Code-submission distributions across task batches.

E Effect of Worker-Queue Backlog on Agents

We select two historical batches under the same runtime conditions and examine a six-hour window after each task starts. The lower-pressure batch records 193 code submissions and the higher-pressure batch records 64; median time to first lease is 5.9 and 31.4 minutes, respectively. As shown in Figure 24, shorter queueing allows an Agent to complete more code submissions under the same execution time and configuration.

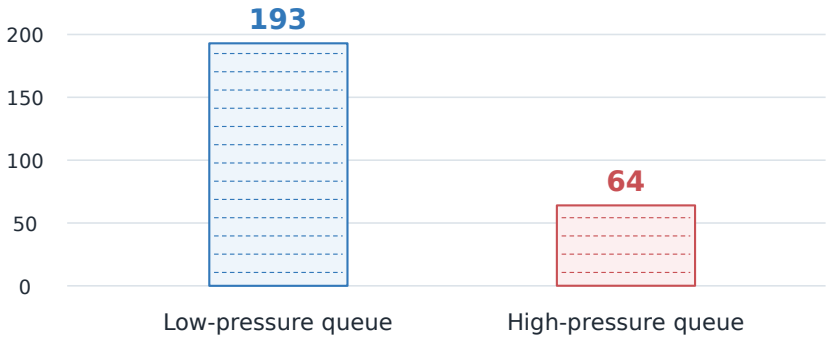


Figure 24: Code submissions within six hours after task start for two historical batches under the same runtime conditions. Blue and red bars denote the lower- and higher-pressure batches; median time to first lease is 5.9 and 31.4 minutes.

F Task Service-Time Distributions across Batches

Figure 25 compares code submissions from the four operator-generation batches in the main text, binned by Worker service time. We include complete records for 9,437 code-under-test items, with 2,066, 4,572, 2,366, and 433 items in the four batches. Service time sums each item's Worker lease-to-release intervals and excludes queueing and retry gaps.

Within each batch, service times are concentrated in a central range with a right tail, while means and variances differ across batches. The horizontal axis is logarithmic, so the histogram shape alone does not imply a normal distribution. These differences may relate to Harness, model, or operator mix, but the figure does not isolate causal effects. For Task Batches A–D, medians are 31.0, 71.6, 34.1, and 11.3 s; means are 54.7, 278.7, 67.9, and 34.0 s; and variances are 19,540.9, 1,154,706.6, 74,045.1, and 5,365.2 s^2, respectively. Variance uses the complete records included for each batch with denominator n.

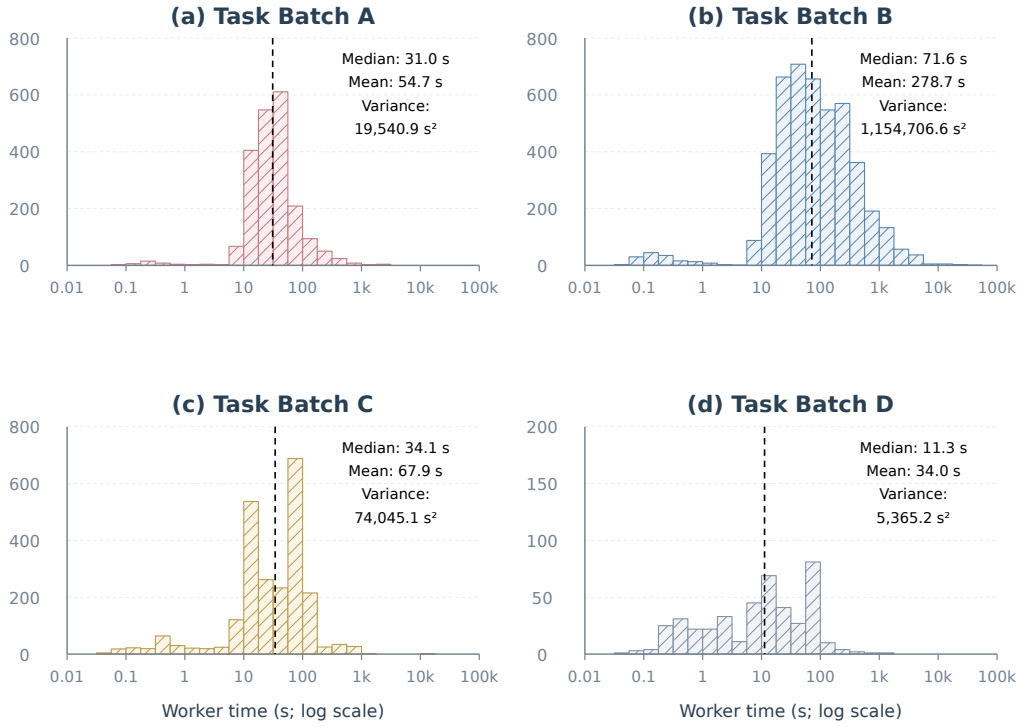


Figure 25: Worker service-time distributions for code submissions from four task batches. The vertical axis is submissions per service-time bin; the horizontal axis is seconds with shared logarithmic bins. Dashed lines mark medians, and the upper-right annotation in each panel reports the mean and variance computed from raw seconds.

G Scheduling Agent Prompt Example

Below is a condensed Scheduling Agent prompt. The Scheduling Agent typically carries out each task in the prompt in sequence.

Scheduling Agent prompt

Role. You are a Controller Agent responsible for monitoring system state and proposing scheduling actions. First read the system description and current runtime state, and identify the available scheduling, task, and service-check tools. Complete the following tasks in order; do not directly modify code, configuration, or task artifacts. Every action must be based on the current snapshot, and the final response must state its evidence.

1. Maintain GPU utilization. For each hardware–runtime queue, when tasks are available, keep the queue near 0–12 items and GPU utilization at 95%–97%. If utilization remains low for about 15 minutes outside batch startup, slightly increase the relevant API concurrency. If the API is overloaded, rate-limited, or disconnected, reduce concurrency, setting it to zero temporarily if needed. When a queue has a clear backlog, do not increase concurrency; defer new leases for 10–20 minutes, or up to 30 minutes in severe cases, and resume proactively after the backlog recedes. Pausing affects only tasks that have not started.

2. Update batch analysis. Select at most three batches from the overview, prioritizing batches never analyzed or analyzed least recently, and skip batches whose analysis finished after stopping. Check stage outputs, error records, and available task logs. Count tasks as completed, failed, incomplete after stopping, running, queued, or uncertain; these states must be mutually exclusive and sum to the task total. Record stage, compilation, and accuracy results separately. For stopped batches, distinguish work completed or failed before stopping from work left incomplete because of stopping. Match multi-configuration batches by their full configuration without omissions or duplicates. Explain whether queued tasks await API concurrency, resources, or another cause, and retain the strongest primary evidence.

3. Clean residue. Check for useless large files left by completed, canceled, or failed tasks, such as profiling databases, dumps, and crash dumps. Clean only residue confirmed to be unused. If files are deleted, report the count, evidence, and confidence; otherwise do not add a cleanup action.

4. One-time admission. If queued tasks are confirmed, a matching online GPU Worker remains below its target utilization, and tasks cannot be leased only because of an API soft-concurrency limit, request one-time admission with a limited validity period and verify that the API hard limit is not exceeded. Do not bypass concurrency limits otherwise.

5. Check automatic recovery. Inspect tasks with automatic recovery enabled and their error logs. If failure is confirmed to result from sending an image to a model without multimodal support, add the corresponding constraint to the recovery prompt to prevent another image submission. For other causes, record evidence without changing recovery behavior by guesswork.

Output. Return this round's observations, actions, and evidence. State uncertainty when evidence is insufficient, and do not invent causes or statistics.