AOS-BENCH: CAN LLM AGENTS MANAGE DYNAMIC WORKLOADS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are increasingly asked to handle multiple requests on the same computer, yet current benchmarks evaluate tasks one at a time. We introduce AOS-Bench, a benchmark that evaluates a persistent agent system as a workload manager: tasks arrive over time and must be delivered under shared CPU and memory limits. Across five model–harness pairs, isolated task success poorly predicts workload delivery. On medium workloads, systems differ by 13 percentage points in isolated success but 29 points in stream success; on heavy workloads, two systems with identical isolated baselines deliver 72.2% and 49.7% of the same tasks. Execution-policy interventions trace much of this gap to task intake: serial dispatch leaves 73% of heavy-workload tasks unclaimed. Yet more elaborate management does not necessarily help—a fixed concurrency of two matches explicit management guidance on medium workloads, while no tested policy uses more than 30% of the provisioned CPU capacity. These results identify workload management as a distinct challenge for agents and establish AOS-Bench as a way to measure it.

1 INTRODUCTION

Advances in coding and computer-use agents let users delegate a growing range of computer tasks through natural language. Benchmarks such as Terminal-Bench and OSWorld typically use individual tasks as the unit of evaluation (Merrill et al., 2026; Xie et al., 2024). New requests may arrive before earlier work is finished, such as file processing requested during a software build. We call the collection of requests handled in a shared computing environment over time a *workload*. In a task-by-task workflow, users launch separate agent sessions and remain responsible for coordinating their work. We envision delegating this responsibility to a persistent agent system that manages an evolving workload on the user's computer.

The agent must construct each task's execution plan while deciding when to accept more work, how to delegate it, and which activities to overlap. Task solving and workload management are therefore coupled. Overlapping execution can use resources left idle during model calls or tool waits, but can also introduce contention and coordination overhead under shared CPU and memory constraints. Performance on isolated tasks does not establish how well an agent maintains progress across requests, or whether its execution decisions improve workload delivery and efficiency.

This motivates our central question: *Can current frontier agent systems effectively manage dynamic workloads under shared resource constraints?* We call this role *Agent as a Workload Manager*. Figure 1 illustrates the role through a limited analogy with an operating system organizing applications.

We introduce AOS-Bench[1], which makes a complete workload the unit of evaluation. A persistent agent system handles tasks from existing CLI benchmarks in a shared environment under fixed CPU and memory limits. To distinguish workload management from task execution, we evaluate each system in two settings: *individual* (each task runs alone, providing an isolated capability baseline) and *stream* (the same tasks arrive over time in a shared environment). We compare verified delivery and execution efficiency across the two settings.

Our preliminary evaluation of five model–harness pairs shows that individual task-solving ability does not predict workload delivery: on the medium tier, systems span only 13 points in individual SR

[1]The name AOS-Bench reflects the vision of an agent serving as a "natural-language operating system" for managing users' computational workloads.

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

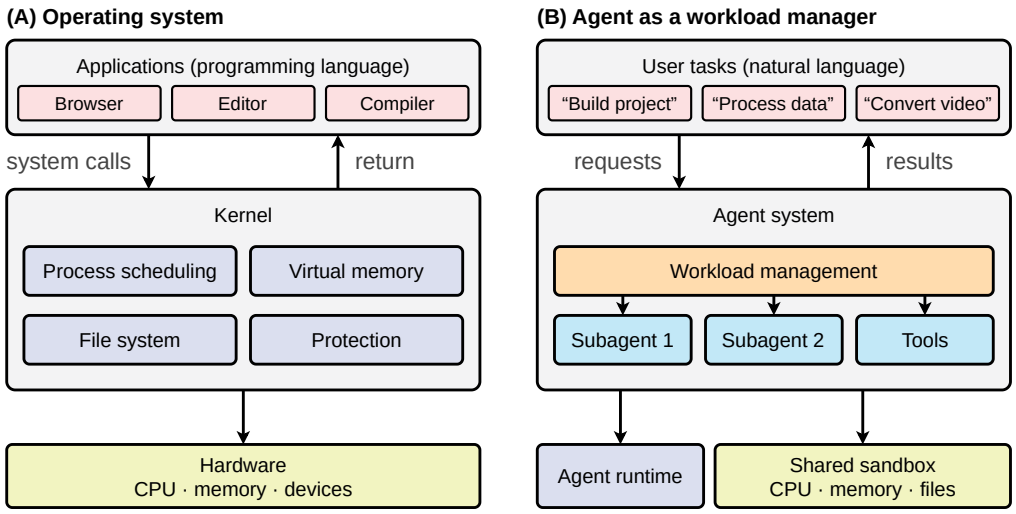

Figure 1: From programs to user tasks. (A) Applications written in programming languages issue system calls to an operating-system kernel, which provides process scheduling, virtual memory, file-system, and protection services over shared hardware. (B) User tasks expressed in natural language are submitted as requests to an agent system. Its workload management delegates work to subagents and tools, and the system runs on an agent runtime and a shared sandbox with CPU, memory, and files. “Natural-language operating system” is a metaphor for this job-management role. AOS-Bench evaluates workload management and task execution.

but 29 points in stream SR, with no system holding the same rank in both orderings. The sharpest case is a pair of systems with an identical individual baseline on heavy workloads that deliver 72.2% and 49.7% on the same task set. Execution-policy interventions locate the mechanism in task intake rather than per-task quality: serial dispatch leaves 73% of heavy-tier tasks unclaimed while native management claims nearly all of them, and trial-level attribution finds task-solving failures in at most three trials per system, so the delivery gap is carried by orchestration outcomes. Managing a workload well, however, is not the same as managing it cleverly: a fixed concurrency of two — a constant with no workload reasoning — matches an agent given explicit management guidance on medium workloads, guidance raises the share of tasks claimed on heavy workloads without raising delivery, and no policy exceeds 30% utilization of the four provisioned cores. A thirty-task case study shows a single system spanning the full outcome range across identical repeats, arguing for treating single-task-at-a-time execution as a stochastic failure mode.

Our contributions are:

- **A formulation of agent workload management** and the AOS-Bench benchmark with construction pipeline, execution protocol, and metrics.
- **A comparative evaluation** of five model–harness pairs establishing that delivery under a shared workload is close to independent of isolated task-solving ability.
- **Execution-policy interventions** identifying task intake as the driver of delivery gaps, and showing that avoiding serial execution captures most of the available gain while dynamic management yields no measurable advantage over a fixed concurrency constant.

2 Evaluation Framework

AOS-Bench evaluates how well an agent system completes a *workload*—a set of tasks released over time under a shared CPU and memory budget. An *agent system* is a model paired with an agent harness that provides its interaction loop, tools, and execution interface. We do not prescribe internal organization: the system may execute requests in its main session, launch subagents, or invoke other tools.

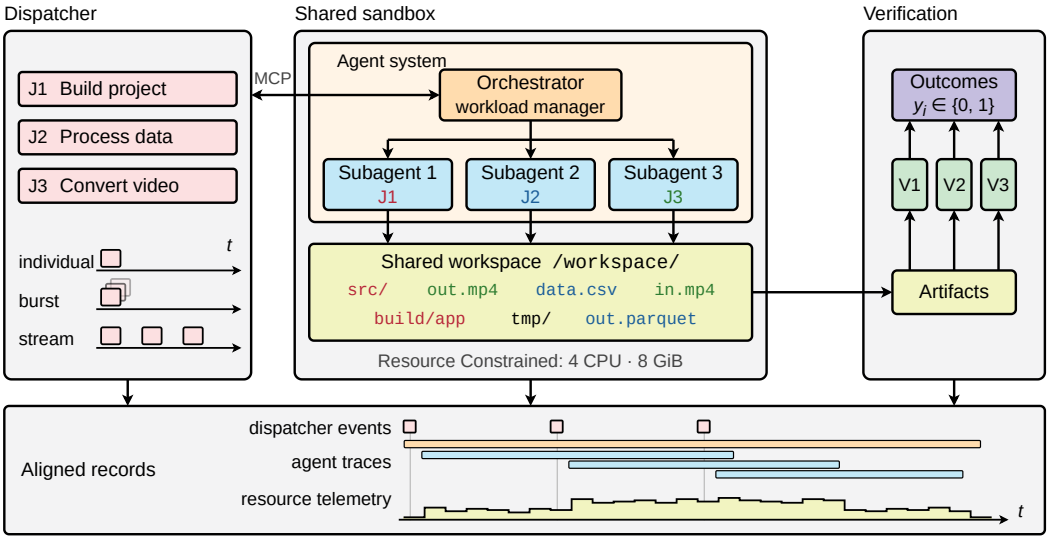

Figure 2: Evaluation framework. An external dispatcher releases tasks at once (*burst*) or over time (*stream*); the *individual* baseline executes each task separately. Through MCP, the agent system can inspect all released tasks, claim work, and delegate execution within a shared 4-CPU, 8 GiB workspace. After execution ends, independent verifiers score final artifacts; dispatcher events, agent traces, and telemetry share a common clock. The illustrated delegation topology is not prescribed.

2.1 Task formulation

A workload is $\mathcal{W} = (\mathcal{J}, B)$ where $\mathcal{J} = \{J_i\}_{i=1}^{n}$ and each task $J_i = (x_i, a_i, V_i)$ consists of a natural-language instruction x_i, a release time a_i, and a verifier V_i that checks success against fixed criteria. B specifies the shared CPU and memory budget; one end-to-end run is a *trial*.

Our primary setting is *stream execution*, in which tasks arrive while the agent is already working. An external dispatcher releases each task J_i at its release time a_i; the agent system cannot see an instruction before its release, nor the schedule of future releases. Through a Model Context Protocol (MCP) interface, the agent system can query the dispatcher to view all tasks released so far and their current states. Using this view, the agent system decides when to take on additional work, how to organize and delegate execution, and when to check progress. A task is *claimed* once the agent system marks it as taken in the dispatcher; a claim only declares ownership and does not indicate that execution has started or succeeded.

We use two comparison settings in addition to stream execution. *Individual* runs execute each task separately and provide a reference for isolated task-solving capability. Because they collectively receive more resource opportunity than a shared run, they serve as a capability reference rather than an equal-cost throughput baseline. *Burst* runs use the same shared environment and task set as stream runs, but release all tasks at $t = 0$. In stream and burst runs, the agent system may still delegate each task to its own subagent, as in individual execution; the difference is that it must create and coordinate these subagents itself within the shared budget. Figure 2 illustrates the framework and its execution settings.

The protocol requires agent and worker execution to stop before verification. The evaluator then runs every task verifier V_i against the final state of the shared workspace. Verification feedback is not exposed during execution. Agent-reported `completed` and `failed` states are retained as behavioral records but do not determine correctness. The default protocol scores all declared tasks, including tasks that were never claimed or never reported `completed`; the preliminary results apply the explicit grader-failure exclusions described in Section 3.1. We additionally record task events, agent and tool traces, and container resource telemetry to characterize how the workload was executed.

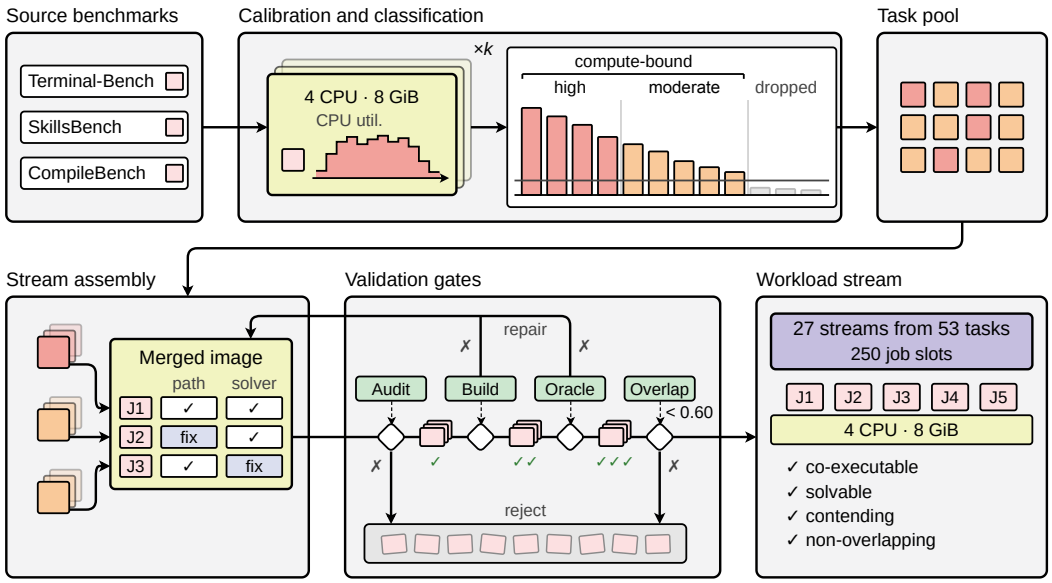

Figure 3: Workload construction pipeline. Source tasks are profiled for resource intensity (L2 moderate, L3 high), assembled into a shared Docker image with consistent path resolution, and admitted only after both image validity and reference-solution success are confirmed. Full selection rules are in Appendix A.

2.2 Workload Construction

Figure 3 summarises the construction pipeline. Source benchmarks include Terminal-Bench (Merrill et al., 2026), SkillsBench (Li et al., 2026), TerminalWorld (Chu et al., 2026), WildClawBench (Ding et al., 2026), CompileBench (Quesma, 2025), CompileAgent (Hu et al., 2025), ParEval (Nichols et al., 2024), and the One Billion Row Challenge (Morling, 2024), supplemented by synthetic tasks. Tasks are standardised in the Harbor format (Harbor Team, 2026) with instructions, environments, reference solutions, and verifiers. Calibration runs measure execution duration and CPU utilization; L2 and L3 denote moderate and high resource intensity (Appendix A).

Task sets are merged into a shared Docker image with consistent path resolution across instructions, reference solutions, and verifiers. Iterative validation applies two gates: a *Docker validity* gate (image builds and launches) and a *solution validity* gate (every reference solution reaches full score in the merged environment). Failed candidates undergo repair and rechecking; admission requires passing all checks.

2.3 Evaluation Metrics

We report two main metrics, task success rate (SR) and the rewarded-task throughput ratio, which measure how well an agent system delivers verified work and how fast it does so relative to individual execution. We also report total execution time (ET) and the memory-allocation overrun rate (labeled OOM in tables) as complementary resource metrics; lower ET reflects an efficiency gain only at comparable SR, since an agent system also finishes faster by leaving tasks unfinished.

Task success rate (SR). SR is the average reward across the scored tasks of a workload:

$$\mathrm{SR}(\mathcal{W}) = \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} y_i, \tag{1}$$

where $\mathcal{I}$ is the set of scored tasks, $|\mathcal{I}|$ is its size, and $y_i \in [0, 1]$ is the reward from task i's verifier, with $y_i = 1$ for full success, $y_i = 0$ for no credit, and intermediate values for partial credit. By default, $\mathcal{I}$ contains all n tasks, including those never claimed or never finished; tasks whose verifier fails to execute are excluded rather than scored as zero (Section 3.1). Tables show $100\,\mathrm{SR}$ and

$\Delta\mathrm{SR} = 100(\mathrm{SR}_{\mathrm{str}} - \mathrm{SR}_{\mathrm{ind}})$; ΔSR is an absolute difference between percentage-valued scores. Seven tasks with evidenced construction defects are excluded from both settings (Appendix E); stream SR also excludes trials with a recorded memory sample above the 8 GiB allocation.

Rewarded-task throughput ratio. The throughput ratio measures how much rewarded work per minute an agent system delivers when it manages a stream, relative to executing the same tasks one at a time. Execution time (ET) is the wall-clock duration of the agent execution phase. it covers model calls, tool use, coordination, and waiting for arrivals, excluding setup and post-run verification. Stream ET is the shared run's elapsed time; the individual reference sums per-task execution times, including failed attempts, for sequential execution. Let $Q_s = \sum_w \bar{K}_{s,w} / \sum_w \bar{T}_{s,w}$ for setting $s \in \{\mathrm{ind}, \mathrm{str}\}$, where $\bar{K}_{s,w}$ and $\bar{T}_{s,w}$ are workload w's total reward and ET averaged over repeats. The throughput ratio is

$$R_{\mathrm{tput}} = \frac{Q_{\mathrm{str}}}{Q_{\mathrm{ind}}} = \frac{\mathrm{SR}_{\mathrm{str}}}{\mathrm{SR}_{\mathrm{ind}}} \cdot \frac{\overline{\mathrm{ET}}_{\mathrm{ind}}}{\overline{\mathrm{ET}}_{\mathrm{str}}}, \tag{2}$$

where SR is occurrence-weighted over the same scored task mixture and $\overline{\mathrm{ET}}_s$ is mean ET per workload. The ratio combines relative reward with relative execution time: 80% reward in 100 minutes versus 60% in 40 minutes gives $0.75 \times 2.5 \approx 1.9$. A value above one indicates more rewarded work per minute than the system's own sequential individual reference. We aggregate rewards and times before dividing, rather than averaging workload ratios (Appendix B).

OOM rate. The Out-of-Memory (OOM) rate is the fraction of stream trials in which any recorded memory sample exceeds the 8 GiB allocation. Such trials are excluded from stream SR calculations, and their fraction is reported separately.

Matched aggregation. Individual SR is weighted by each task variant's occurrence frequency in the stream workloads so both sides score the same task mixture. Metrics are first averaged within workloads, then equally across workloads; repeating a workload does not increase its weight.

3 Experimental Results

3.1 Setup

Systems and tiers. We evaluate five model–harness pairs: Claude Opus 5 (Anthropic, 2026b), GLM-5.3 (Zhipu AI, 2026), and DeepSeek-V4-Pro (DeepSeek-AI, 2026) with Claude Code (Anthropic, 2026a); GPT-5.6-sol (OpenAI, 2026a) with Codex (OpenAI, 2026b); and Kimi-K3 (Moonshot AI, 2026b) with Kimi Code (Moonshot AI, 2026a). Both Claude Code and Codex use the `high` reasoning-effort setting (versions in Table 2). We report medium (10 tasks, 10 workloads) and heavy (20 tasks, 5 workloads) tiers; a thirty-task workload is used only as a case study (Section 4.3).

Table 1: Workload-tier definitions. Outer limit is a harness termination, not a task deadline.

Tier	Tasks	L2	L3	n	Limit (min)
Medium	10	10	0	10	150
Heavy	20	10	10	5	240

Execution and scoring. Each run is allocated 4 CPUs and 8 GiB, shared in stream and exclusive in individual execution. We target three attempts per workload and per task and retain the first three eligible repeats under the *full-clean* filter: reward record, transcript, $\geq$8 trajectory steps, and no output-cap, throttling, dead-route, or verifier-fetch failure; full criteria in Appendix H. Individual SR is weighted by each task's stream-occurrence frequency so both sides score the same mixture; task matching uses source identifier and instruction SHA-256 hash. Seven tasks with evidenced construction defects and three with broken graders are excluded from both settings (Appendix E); retained scoring sets contain 95 and 98 task occurrences in medium and heavy. GPT-5.6-sol medium stream is incomplete at 27 of 30 trials due to an image-level setup failure (Appendix H.7). CPU estimates come from cumulative cgroup counters aligned to execution-phase timestamps; OOM trials (memory exceeds 8 GiB allocation) are excluded from stream SR. SR differences are paired and use a cluster bootstrap over workloads (4000 resamples); medium has ten clusters and heavy five.

Table 2: Individual and stream results. SR is mean per-task reward; Δ = Str. − Ind. (%); individual SR is occurrence-weighted and matched by task id and instruction hash. Seven construction-defective tasks excluded from both sides; stream trials exceeding the 8 GiB allocation excluded from Str. SR (OOM column = % excluded). Throughput = rewarded-task rate vs. sequential individual; bold = best per tier for Δ and Throughput.

Model	Harness	SR (%)			Throughput	ET (min)		OOM (%)	
		Ind.	Str.	Δ	ratio (×)	Ind.	Str.	Ind.	Str.
Medium (10 workloads, 20 minutes of arrivals, 150-minute limit)									
Claude Opus 5	Claude Code	89.7	85.2	**−4.6**	**2.46**	240.7	94.8	0.0	3.3
Kimi-K3	Kimi Code	79.6	69.4	−10.2	2.34	314.2	122.4	0.0	3.7
GPT-5.6-sol	Codex	92.8	71.7	−21.1	1.71	136.0	63.9	0.0	0.0
GLM-5.3	Claude Code	81.7	56.3	−25.5	1.88	390.7	148.8	0.0	0.0
DeepSeek-V4-Pro	Claude Code	87.1	57.0	−30.1	1.56	340.4	147.4	0.0	0.0
Heavy (5 workloads, 60 minutes of arrivals, 240-minute limit)									
Claude Opus 5	Claude Code	72.4	72.2	**−0.2**	3.16	477.9	171.6	2.2	20.0
DeepSeek-V4-Pro	Claude Code	53.9	36.1	−17.8	2.38	807.1	240.1	0.0	6.7
Kimi-K3	Kimi Code	64.9	46.6	−18.3	2.87	708.2	213.6	2.2	40.0
GLM-5.3	Claude Code	68.4	49.4	−19.0	**3.60**	916.6	217.8	9.3	20.0
GPT-5.6-sol	Codex	72.4	49.7	−22.6	1.41	343.0	174.2	2.2	0.0

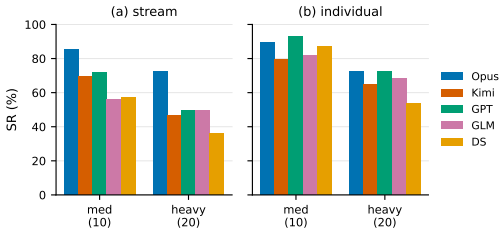

Figure 4: Mean per-task reward for the five systems under individual and stream settings. Arms: opus = Claude Opus 5, gpt = GPT-5.6-sol, ds = DeepSeek-V4-Pro; tiers are medium (10 tasks) and heavy (20 tasks), with disjoint task sets. Comparisons are within-tier only; the individual setting uses no scheduling preamble (Appendix H.5).

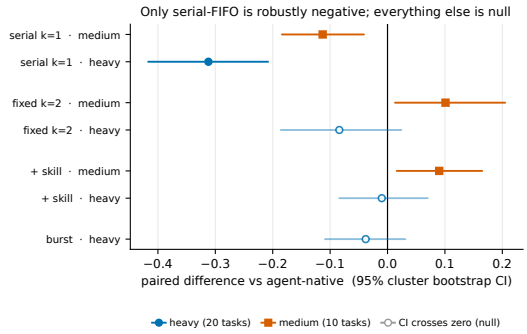

Figure 5: How each execution policy changes GLM-5.3's SR relative to native management under stream arrivals. Each point is the policy's SR minus native SR (as a fraction), with a 95% bootstrap interval over workloads; filled markers do not cross zero. "+ skill" is the guidance condition in Table 3. Differences are paired by task and omit tasks that no run solves, so they differ slightly from row differences in Table 3.

3.2 Main Results

Table 2 reports frequency-matched individual and stream SR alongside the rewarded-task throughput ratio and total execution time (ET) for the three workload tiers. Individual ET is the sum of per-task execution times averaged across workloads; stream ET is the elapsed shared execution interval.

How much single-task ability survives contact with a workload separates the systems more than the ability itself. On heavy workloads Claude Opus 5 delivers 72.2% against a 72.4% individual baseline ($\Delta = -0.2$), whereas the other four lose 17.8 to 22.6 points (retaining 67%–72% of their baselines); on medium Opus loses 4.6 points (95% retention) and the others 10.2 to 30.1 (65%–87%). The spread is not a spread in task-solving ability. GPT-5.6-sol has the highest individual baseline in both tiers (92.8% and 72.4%), and on heavy both GPT and Opus share the same 72.4% individual baseline — yet GPT delivers 49.7% against Opus's 72.2% on the identical task set, a gap of 22.5 points that appears only under shared execution. DeepSeek-V4-Pro shows the same pattern more

sharply: a medium baseline of 87.1%, second only to GPT-5.6-sol, yet the tier's lowest stream SR at 57.0%. Individual ability therefore does not determine workload delivery, and the two are close to independent here: the medium baselines span 13.2 points across the five systems while their stream scores span 28.9 points, and no system holds the same rank in both orderings on that tier. The quantity that separates the systems is what they do with the workload, not what they can do with a task. Appendix D traces the heavy-tier gap to a specific behaviour: the systems that lose the most stop claiming released tasks and execute one at a time, leaving 25% and 48% of released tasks never claimed against 0% for Claude Opus 5.

Only serial dispatch reliably lower SR. Figure 5 checks which of these policy differences exceed workload-to-workload variation. Serial FIFO is the only policy that lowers SR on both tiers, and its cost grows with workload size: -11.3% on medium and -31.2% on heavy, with neither interval crossing zero. On medium, fixed $k = 2$ and guidance improve on native ($+10.1\%$ and $+9.0\%$); on heavy, no other policy's interval excludes zero, though five workloads make this a coarse check. These paired estimates omit tasks no run solves, so they differ from the row differences in Table 3 but preserve the policy ordering.

Repeated instances bias stream scores upward. Workloads contain multiple instances of the same task sharing one filesystem, so an artifact one instance produces can satisfy a sibling's verifier; isolated execution cannot do this. This biases stream SR upward wherever families occur, so all reported decreases are conservative.

Throughput distinguishes delivery from shorter execution. On medium workloads the rewarded-task throughput ratios are $2.46\times$ for Opus, $2.34\times$ for Kimi, $1.88\times$ for GLM, $1.71\times$ for GPT-5.6-sol, and $1.56\times$ for DeepSeek. All five exceed unity, so every system delivers rewarded work faster than sequential individual execution even where its per-task reward drops. The ordering is not the ordering of stream SR: GPT-5.6-sol has the tier's second-highest stream SR but the second-lowest throughput ratio, because its own sequential reference already delivers rewarded work at the highest rate in the tier, so the relative gain is smaller. The ratio therefore measures delivery rate against a system's own sequential reference, not cross-system speed. On heavy the ratios rise for four of five systems, to between $2.38\times$ and $3.60\times$, while GPT-5.6-sol falls to $1.41\times$.

Execution time and memory demand. Stream execution reduces per-workload execution time (ET) by factors of 1.4–4.6 across all model–tier combinations. These reductions do not consistently track changes in SR: GLM and DeepSeek achieve substantial time savings while also showing sizeable SR declines. Shorter ET therefore does not necessarily indicate an efficiency gain at comparable output quality. Recorded memory-allocation overruns are more frequent on heavy workloads, affecting 20.0% of Opus and GLM stream trials and 40.0% of Kimi stream trials, compared with near-zero rates in individual execution. Trials with a recorded memory sample above the 8 GiB allocation are excluded from stream SR; their frequency is reported separately (Appendix H).

4 Analysis

4.1 Execution organization and workload delivery

We compare GLM-5.3 with Claude Code under four execution policies: (i) native agent management, (ii) native with additional resource and intake guidance, (iii) serial FIFO ($k = 1$), and (iv) fixed concurrency of two tasks ($k = 2$). Fixed policies use the same model and solver but dispatch via a driver; task sets, arrival schedules, and resource limits are matched.

4.2 Arrival conditions

The GLM heavy burst comparison attains 48.8% SR against 50.7% for stream arrivals, with completion shares of 58.3% and 56.3%; the 1.9-point difference is smaller than the trial-to-trial spread, so neither an arrival-condition effect nor equivalence is established. Claim latency separates sharply (349 s burst vs. 1187 s stream), yet the same system converts a 3.4-fold reduction in intake delay into no measurable SR change, indicating that delay is not the binding constraint once tasks are available.

Table 3: GLM-5.3 execution-policy comparisons under stream arrivals. k limits unfinished solver sessions; t_{claim} is median seconds from claimable to claimed.

Tier	Policy	N	SR (%)	ET (min)	Claim (%)	Compl. (%)	t_{claim} (s)	CPU (min)
Med	Serial (k=1)	30/30	41.1	151.8	50.0	39.3	3480	99.8
Med	Native	30/30	52.8	148.8	77.0	63.3	1213	103.9
Med	Guidance	30/30	57.1	149.4	80.3	66.7	1785	131.4
Med	Fixed (k=2)	30/30	58.1	150.0	80.3	60.7	2893	163.3
Hvy	Serial (k=1)	15/15	25.4	240.0	26.7	21.7	4359	156.8
Hvy	Native	15/15	50.7	217.8	75.0	56.3	1187	282.7
Hvy	Guidance	15/15	50.1	219.8	81.0	56.0	2691	246.6
Hvy	Fixed (k=2)	15/15	43.6	240.0	50.0	39.3	3795	269.1

Delivery follows intake, not per-task quality. Serial FIFO has the lowest SR (41.1%/25.4%) and completion share (39.3%/21.7%), with claim latency roughly 3× native; on heavy 73.3% of tasks are never claimed.

Fixed concurrency matches on medium, fails on heavy. Fixed $k = 2$ matches the guided condition on medium (5.3% above native) but drops to 43.6% on heavy — a constant tuned for 10 tasks does not transfer to 20.

More intake need not mean more delivery. Guidance raises coverage on both tiers but SR only on medium; on heavy SR is unchanged within 0.6% while coverage rises 6 points. Serial heavy at a 10-hour limit recovers 56% of the gap, confirming the policy is budget-bound as well as intake-bound.

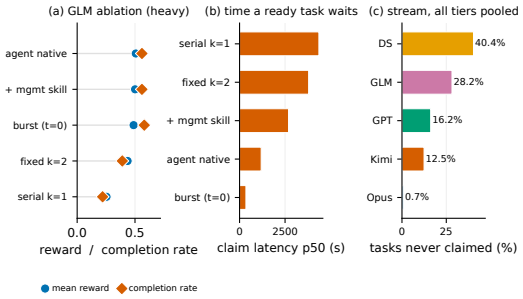

Figure 6: (a) SR and completion share for the four GLM-5.3 policies. (b) Median claim latency. (c) Never-claimed share per system, pooled over all stream tiers.

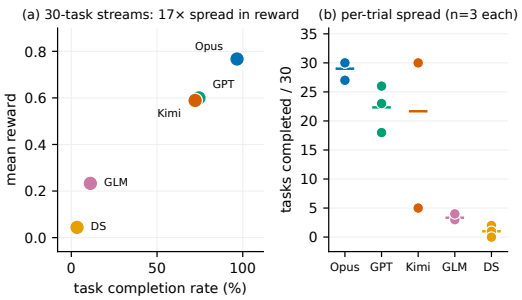

Figure 7: Thirty-task case study. (a) Mean reward vs. completion rate. (b) Per-trial completion counts (three repeats per system). Kimi-K3 spans 30, 30, and 5 completions across identical repeats.

4.3 A THIRTY-TASK WORKLOAD AS A CASE STUDY

A thirty-task workload, three repeats per system, stresses intake harder than either reported tier and exposes the mechanism in isolation. Stream SR spans 4.4% to 76.7% against individual baselines of 54.4% to 71.7%. Opus claims 87 of 90 possible task instances; GPT and Kimi complete 67 and 65; GLM completes 10 and DeepSeek 3, both holding never more than one task at a time and reaching the 360-minute limit with 77 and 84 instances left unclaimed. Both systems score 63–64% on the individual baseline, so the deficit is not inability to solve the tasks.

Serial execution varies across repeats. Kimi-K3 completes 30, 30, and 5 tasks in three repeats under the same workload and configuration. In the third repeat, it holds only one task at a time, resembling the GLM and DeepSeek trajectories; in the other two, it claims the full workload. The same system thus exhibits both execution regimes, suggesting that serial execution is an intermittent failure mode rather than a fixed limit on its task-solving capability. Opus shows more consistent intake, claiming 87 of 90 task instances across the three repeats.

Trajectory evidence and terminal-state decompositions are in Appendix I and J.

5 LIMITATIONS

The evaluation setting bounds what our results can mean. A task consists of an instruction, a release time, and a verifier, with no deadline, no value or priority, and no dependency on another task. The dispatcher releases work first-in, first-out, so a system cannot choose which released task to serve next. We therefore measure intake rate and execution organization, but never the ordering policy that scheduling theory treats as the central decision, and because every task is worth the same, the

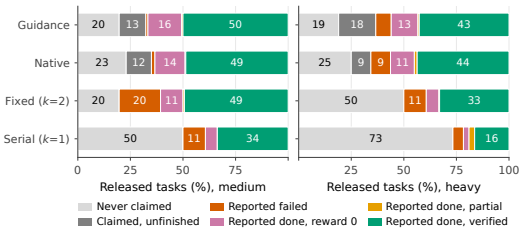

Figure 8: Terminal states of released tasks for the four GLM-5.3 policies. Greys: never served; warm: no credit; green: verified.

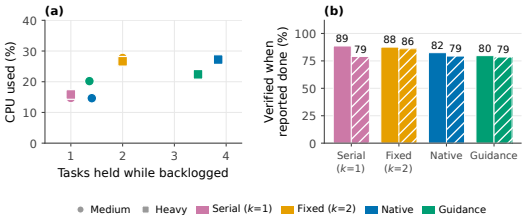

Figure 9: (a) Tasks held vs. CPU utilization; (b) fraction of reported-complete tasks verified. No policy exceeds 30% CPU; agent-managed policies have lowest verification accuracy.

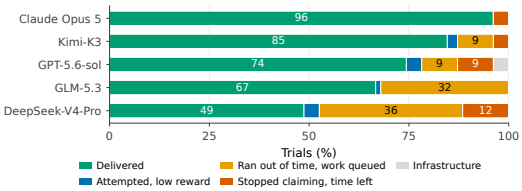

Figure 10: Trial disposition across 390 native-stream trials. Attempted-and-failed is rare ($\leq$3 per system); orchestration failures carry the delivery gap.

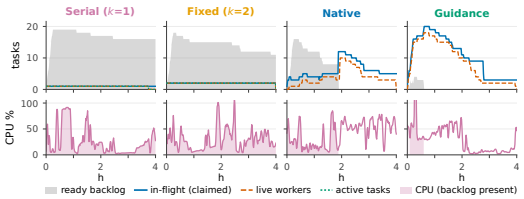

Figure 11: One heavy workload under the four policies on a shared clock. Serial and fixed k=2 leave 16 and 11 of 20 tasks unclaimed; agent-managed policies drain the backlog within two hours.

objective reduces to maximizing verified count under a shared budget. The stream and individual settings also differ in more than scheduling. The stream setting adds a shared filesystem, a task-pool interface, a scheduling preamble, and contention for one allocation, so the paired difference measures the cost of moving a task from isolation into a managed workload rather than the cost of scheduling decisions alone. Individual runs additionally receive more aggregate resource opportunity, which makes them a capability reference rather than an equal-cost baseline. The workload design samples one region of a larger space. Tiers of 10, 20, and 30 tasks over 4 CPUs and 8 GiB draw disjoint task sets, so tier differences confound scale with composition, and one burst condition does not characterize sensitivity to the arrival process. Whether these failure modes persist under tighter resource ratios, or under hundreds rather than tens of concurrent requests, is outside what this design establishes. Inference is finally bounded by workloads rather than trials, since repeats share a task set and an individual baseline. That leaves ten clusters on medium and five on heavy, too few for a calibrated interval, and most heavy trials stop at the wall-clock limit, so delivery on that tier is a lower bound.

6 Conclusion

We introduce AOS-Bench to evaluate agent systems as workload managers, using verified CLI tasks that arrive over time in a shared container. Across five model–harness pairs, isolated task success does not predict workload delivery. On medium workloads, individual success rates span 13.2 points, while stream success rates span 28.9 points; on heavy workloads, two systems with identical individual baselines deliver 49.7% and 72.2% of tasks. Policy interventions show that serial execution misses newly released tasks, while a fixed concurrency of two matches guided management on medium workloads but not heavy ones. Identical repeats also reveal large variation within a single system. These results establish workload management as a distinct evaluation dimension and motivate closer analysis of resource use and execution trajectories.

ETHICS STATEMENT

AOS-Bench uses tasks from existing benchmarks and runs agent-written commands in containers shared by the tasks in each trial. Release requirements include documenting upstream licenses, excluding secrets and private data, and describing the risks of executing untrusted code.

REPRODUCIBILITY STATEMENT

The planned release includes a frozen workload manifest, container and agent image identifiers, task and verifier provenance, prompts, and launcher configurations.

AI USE STATEMENT

The authors used generative AI tools for research planning, feedback on methods and experiments, software development and review, data inspection, and manuscript drafting and editing. The authors reviewed the resulting materials and interpretations. They take responsibility for the methods, results, claims, code, and text.

REFERENCES

Justice Owusu Agyemang, Jerry John Kponyo, Obed Kwasi Somuah, Elliot Amponsah, Godfred Manu Addo Boakye, and Kwame Opuni-Boachie Obour Agyekum. HiveMind: OS-Inspired Scheduling for Concurrent LLM Agent Workloads. *arXiv preprint arXiv:2604.17111*, 2026. URL https://arxiv.org/abs/2604.17111.

Anthropic. Claude Code: Overview. Product documentation, 2026a. URL https://code.claude.com/docs/en/overview. Accessed September 26, 2026.

Anthropic. Claude Opus 5 System Card. System card, 2026b. URL https://www.anthropic.com/claude-opus-5-system-card. Accessed September 26, 2026.

Axel Backlund and Lukas Petersson. Vending-Bench: A Benchmark for Long-Term Coherence of Autonomous Agents. *arXiv preprint arXiv:2502.15840*, 2025. URL https://arxiv.org/abs/2502.15840.

Shoufa Chen, Luyuan Wang, Xuan Yang, Zhiheng Liu, Yuren Cong, Yuanfeng Ji, Feiyan Zhou, Xiaohui Zhang, Fanny Yang, and Belinda Zeng. TUA-Bench: A Benchmark for General-Purpose Terminal-Use Agents. *arXiv preprint arXiv:2606.28480*, 2026. URL https://arxiv.org/abs/2606.28480.

Zhaoyang Chu, Jiarui Hu, Xingyu Jiang, Pengyu Zou, Han Li, Chao Peng, Peter O'Hearn, Earl T. Barr, Mark Harman, Federica Sarro, and He Ye. TerminalWorld: Benchmarking Agents on Real-World Terminal Tasks. *arXiv preprint arXiv:2605.22535*, 2026. URL https://arxiv.org/abs/2605.22535.

DeepSeek-AI. DeepSeek-V4-Pro GA Release. Official release announcement, 2026. URL https://api-docs.deepseek.com/news/news260813/. Accessed September 26, 2026.

Shuangrui Ding, Xuanlang Dai, Long Xing, Shengyuan Ding, Ziyu Liu, Jingyi Yang, Penghui Yang, Zhixiong Zhang, Xilin Wei, Xinyu Fang, Yubo Ma, Haodong Duan, Jing Shao, Jiaqi Wang, Dahua Lin, Kai Chen, and Yuhang Zang. WildClawBench. GitHub repository, 2026. URL https://github.com/InternLM/WildClawBench. Accessed September 24, 2026.

Wei Fan, Xinjie Shen, Xudong Guo, Jianhong Tu, Yang Su, Yinger Zhang, Lianghao Deng, Fengyu Wang, Baohua Dong, Yangqiu Song, and Dayiheng Liu. E-Commerce Bench: Evaluating LLM Agents on Long-Horizon Autonomous Business Operation. *arXiv preprint arXiv:2608.30730*, 2026. URL https://arxiv.org/abs/2608.30730.

Harbor Team. Harbor Documentation: Task Overview. Official documentation, 2026. URL https://docs.harborframework.com/core-concepts/tasks/overview. Accessed September 24, 2026.

Li Hu, Guoqiang Chen, Xiuwei Shang, Shaoyin Cheng, Benlong Wu, Gangyang Li, Xu Zhu, Weiming Zhang, and Nenghai Yu. CompileAgent: Automated Real-World Repo-Level Compilation with Tool-Integrated LLM-based Agent System. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2078–2091. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.103. URL https://aclanthology.org/2025.acl-long.103/.

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*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating Multimodal Agents on Realistic Visually Grounded Web Tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.50. URL https://aclanthology.org/2024.acl-long.50.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, Chujun Tao, Binxu Li, Xuandong Zhao, Hejia Geng, Xiaojun Wu, Junwei Zhou, Xiaokun Chen, Hanwen Xing, Yubo Li, Qunhong Zeng, Di Wang, Yuanli Wang, Roey Ben Chaim, Penghao Jiang, Haotian Shen, Luyang Kong, Xinyi Liu, Runhui Wang, Xuanqing Liu, Jiachen Li, Xin Lan, Yueqian Lin, Wengao Ye, Junwei He, Songlin Li, Yue Zhang, Yipeng Gao, Yijiang Li, Ze Ma, Liqiang Jing, Tianyu Wang, Kaixin Li, Yiqi Xue, Haoran Lyu, Yizhuo He, Yuchen Tian, Shutong Wu, Bowei Wang, Yixuan Gao, Bo Chen, Litong Liu, Sikai Cheng, Jiajun Bao, Shuaicheng Tong, Shuwen Xu, Terry Yue Zhuo, Tinghan Ye, Qi Qi, Miao Li, Longtai Liao, Zelin Tan, Chang Shi, Xilin Tang, Srinath Tankasala, Boqin Yuan, Yaoyao Qian, Jianhong Tu, Chenguang Wang, Yizhou Sun, Wei Wang, Aaron Taylor, Ziyue Yang, Changkun Guan, Zhikang Dong, Xinyu Zhang, Steven Dillmann, Han-chung Lee, and Dawn Song. SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks. *arXiv preprint arXiv:2602.12670*, 2026. URL https://arxiv.org/abs/2602.12670.

Kai Mei, Xi Zhu, Wujiang Xu, Mingyu Jin, Wenyue Hua, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. AIOS: LLM Agent Operating System. In *Proceedings of the Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=L4HHkCDz2x.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Moonshot AI. Kimi Code. Product documentation, 2026a. URL https://www.kimi.com/code/en. Accessed September 26, 2026.

Moonshot AI. Kimi K3. Model card, 2026b. URL https://huggingface.co/moonshotai/Kimi-K3. Accessed September 26, 2026.

Gunnar Morling. The One Billion Row Challenge. GitHub repository, 2024. URL https://github.com/gunnarmorling/1brc. Accessed September 24, 2026.

Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can Large Language Models Write Parallel Code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. Association for Computing Machinery, 2024. URL https://pssg.cs.umd.edu/assets/papers/2024-06-pareval-hpdc.pdf.

OpenAI. GPT-5.6 Sol. Model documentation, 2026a. URL https://developers.openai.com/api/docs/models/gpt-5.6-sol. Accessed September 26, 2026.

OpenAI. OpenAI Codex CLI. GitHub repository, 2026b. URL https://github.com/openai/codex. Accessed September 26, 2026.

Yijun Pan, Yukun Lian, Kunyu Shi, Junbo Li, Hongwei Xue, Sicong Xie, Guannan Zhang, and Xiaoying Xing. Business Arena: Benchmarking LLM Agents in a Realistic Marketplace. *arXiv preprint arXiv:2608.08621*, 2026. URL https://arxiv.org/abs/2608.08621.

Quesma. CompileBench. GitHub repository, 2025. URL https://github.com/QuesmaOrg/CompileBench. Accessed September 24, 2026.

Zachary S. Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. CORE-Bench: Fostering the Credibility of Published Research Through a Computational Reproducibility Agent Benchmark. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=BsMMc4MEGS.

Yifan Sui, Han Zhao, Rui Ma, Zhiyuan He, Hao Wang, Jianxun Li, Kaiqiang Xu, Kai Chen, and Yuqing Yang. Parallelizing Tool Execution and LLM Generation for Low-Latency Agent Serving. *arXiv preprint arXiv:2603.18897*, 2026. URL https://arxiv.org/abs/2603.18897.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. In *Advances in Neural Information Processing Systems*, volume 37, pp. 52040–52094, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. TheAgentCompany: Benchmarking LLM Agents on Consequential Real World Tasks. In *Advances in Neural Information Processing Systems*, volume 38, pp. 10291–10321, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/0d744742f6fac4d1134c019b7cef3c8a-Abstract-Datasets_and_Benchmarks_Track.html.

Yihui Zhang, Tianyu Wo, Jinghao Wang, Xiaoyang Sun, Menghao Zhang, Cangzhou Yuan, Li Li, Chunming Hu, Albert Y. Zomaya, and Renyu Yang. SpecBox: Speculative Sandbox Scheduling for Efficient LLM Agent Serving. *arXiv preprint arXiv:2607.23933*, 2026. URL https://arxiv.org/abs/2607.23933.

Yusheng Zheng, Jiakun Fan, Quanzhi Fu, Yiwei Yang, Wei Zhang, and Andi Quinn. AgentCgroup: Understanding and Controlling OS Resources of AI Agents. *arXiv preprint arXiv:2602.09345*, 2026. URL https://arxiv.org/abs/2602.09345.

Zhipu AI. GLM-5. Model documentation, 2026. URL https://docs.z.ai/guides/llm/. Accessed September 26, 2026.

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*, 2024. URL https://openreview.net/forum?id=oKn9c6ytLx.

A Details of Data Constructions

Task sources. Task sources include Terminal-Bench (Merrill et al., 2026), SkillsBench (Li et al., 2026), TerminalWorld (Chu et al., 2026), WildClawBench (Ding et al., 2026), CompileBench (Quesma, 2025), CompileAgent (Hu et al., 2025), ParEval (Nichols et al., 2024), and the One Billion Row Challenge (Morling, 2024). We also include supplementary synthetic tasks developed for AOS-Bench.

Figure 12 gives the resulting composition of the evaluated workloads. Terminal-Bench is the largest contributor in every tier, at 40% to 73% of task occurrences, so the workloads are not balanced across sources and a per-source effect cannot be separated from a per-tier one. The counts are task *occurrences* summed over each tier's streams, which is the work an agent had to serve; the distinct-task counts are lower because streams may release repeated instances of one task, at 21, 34, 48, and 19 distinct tasks for light, medium, heavy, and heavy-plus against 50, 100, 100, and 30 occurrences. Heavy-plus draws on three sources only, being a single workload.

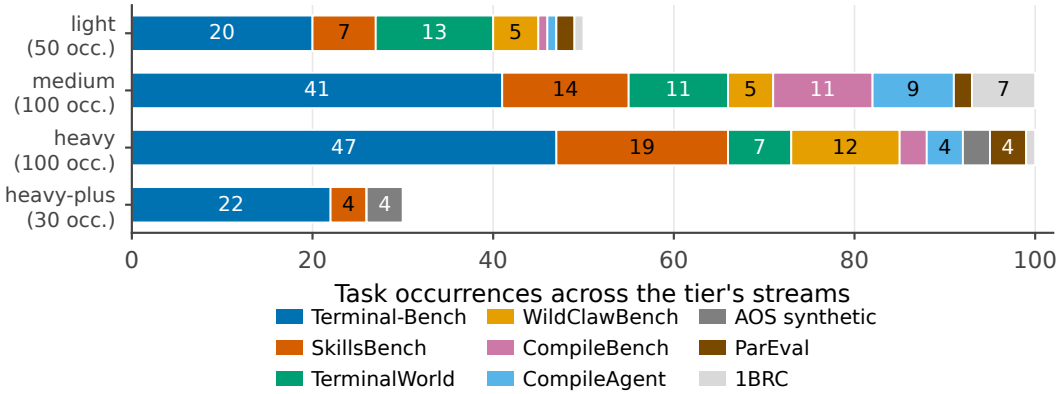

Figure 12: Source-benchmark composition of the 26 evaluated stream workloads, as task occurrences across each tier's streams (10, 10, 5, and 1 workloads). Segments under four occurrences are unlabeled. Occurrences exceed distinct tasks because a stream may release repeated instances of one task.

Task eligibility and calibration. Source tasks must provide instructions, input assets, executable verifiers, and environments compatible with the shared container configuration. The task pool uses two valid calibration runs per task with GLM-5.2 (Zhipu AI, 2026) under 4 CPUs and 8 GiB of memory. Calibration uses a different model from any evaluation system, so tier labels are not tuned to the systems later measured against them. Valid calibration requires usable execution and resource records; it does not imply verification success. Reference-solution validation separately checks solvability after merging.

Resource score and tier boundaries. For task i and calibration run k, let w_{ik} be elapsed time in seconds and u_{ik} be mean CPU utilization divided by the allocated CPU capacity, so that $u_{ik} = 1$ denotes full use of the quota. We compute the resource score as $s_i = \frac{1}{2}\sum_{k=1}^{2} \min(w_{ik}/3600, 1)\min(u_{ik}, 1)$. The classification assigns L1 to $s_i < 0.009$, L2 to $0.009 \leq s_i < 0.10$, and L3 to $s_i \geq 0.10$. L1, L2, and L3 denote low, moderate, and high observed resource intensity, respectively. Workload construction uses tasks from L2 and L3. Tier labels are those the construction pipeline assigned from these thresholds.

Workload overlap. For the task sets $\mathcal{J}_a$ and $\mathcal{J}_b$ of two workloads, we define overlap as $|\mathcal{J}_a \cap \mathcal{J}_b| / \min(|\mathcal{J}_a|, |\mathcal{J}_b|)$ and require it to be below 0.60 against each admitted workload. The smaller-set denominator limits near-containment of smaller workloads in larger ones. This criterion limits task reuse without guaranteeing statistical independence.

A.1 STREAM DATA FORMAT

Our stream format is adapted from the Harbor task format (Harbor Team, 2026). A Harbor task packages an instruction in `instruction.md`, configuration in `task.toml`, an execution environment in `environment/`, a reference solution in `solution/`, and a verifier in `tests/`. We extend this organization by placing task-specific instructions and verifiers under `tasks/<task_id>/`, merging the environments, and replacing the top-level task configuration with `stream.toml`. Figure 13 shows a minimal two-task example; its names and release times are illustrative.

Directory layout

```
example-stream/
|-- stream.toml
|-- environment/
|   |-- Dockerfile
|   `-- _sources/...
|-- tasks/
|   |-- build-project/
|   |   |-- instruction.md
|   |   `-- tests/
|   |       |-- test.sh
|   |       `-- ...
|   `-- process-data/
|       |-- instruction.md
|       `-- tests/
|           |-- test.sh
|           `-- ...
`-- solution/
    |-- solve.sh
    `-- tasks/<task_id>/...
```

`stream.toml`

```
version = "1.0"

[metadata]
name = "example-stream"

[environment]
cpus = 4
memory_mb = 8192
allow_internet = true

[session]
max_steps = 360

[[arrivals]]
task_id = "build-project"
arrive_wall_time = 0

[[arrivals]]
task_id = "process-data"
arrive_wall_time = 60
```

Figure 13: Stream packaging adapted from Harbor tasks. One environment serves both tasks; separate task directories retain their instructions and verification assets. The configuration adds timed task releases to the shared environment specification.

Configuration fields. `version` identifies the format version, and `metadata.name` identifies the workload. The `environment` block retains Harbor's environment configuration fields; its CPU and memory limits apply to the shared container, and `docker_image` may additionally specify a prebuilt image. The `session.max_steps` field stores session-level execution metadata; its enforcement depends on the configured harness. Each `arrivals` entry links a `task_id` to its task directory and an `arrive_wall_time`, the release offset in seconds from the dispatcher's session start. This offset corresponds to a_i in Section 2; setting all offsets to zero produces the burst condition. Tasks have no deadline fields in the evaluation configuration.

Runtime interpretation. The directory tree is a dataset package, not a set of per-task runtime sandboxes. The merged environment initializes a shared `/workspace`; task instructions identify the relevant input and output paths within it. Input files may already exist when the session starts, while the external dispatcher exposes each task only after its release time and allows the agent to inspect all released tasks. The full schedule, reference solutions, and verification assets are evaluator-side data and are not exposed as agent execution inputs. Reference solutions support construction-time validation; after agent execution, each task's `tests/test.sh` checks the final shared workspace. Task-level rewards are collected by `task_id`, preserving the link between each request and its verified outcome.

B CONFIGURATION AND REPRODUCIBILITY

The present evaluation is a preliminary snapshot rather than a frozen release. Table 4 distinguishes recorded design settings from metadata that still require confirmation against actual trials.

Table 4: Recorded configuration and remaining reproducibility checks.

Item	Recorded setting	Required confirmation
Workloads	26: 10 light, 10 medium, 5 heavy	Task IDs, source versions, unique-task count, overlap matrix, and manifest
Task count	5, 10, 20 (30 for case study)	Composition of each evaluated workload
Resources	4 CPUs/8 GiB throughout	Actual cgroup limits, hardware, I/O, and memory units
Arrivals	Stream; additional GLM burst runs	Per-task release times and matched schedules
Repeats	Target of three attempts per workload	Attempt identity, reruns, and duplicate archives
Evaluated systems	`codex`, `glm`, `kimi`, `cc`	Model IDs, harness versions, reasoning settings, and prompts
Execution limit	Historical coupling to outer request timeout; four-hour cutoffs in heavy runs	Independent agent budget, process termination, and verifier budget
Concurrency	Harness-specific capabilities and defaults	Worker limits, asynchronous tool behavior, and configuration overrides
Service settings	GLM output-cap revisions and service throttling	Output limits, endpoint configuration, retry policy, and availability
Telemetry	Design specifies five-second cgroup samples	Actual intervals, missing data, timestamps, and trace completeness
Scoring	Raw task rewards from verifier outputs	Reward schema, full-success thresholds, and status-conditioned fields

Final statistical reporting should first aggregate repeats within each workload, then macro-average across workloads. Comparisons should use common tasks or workloads and state exclusions and their reasons. Repeated attempts of one workload are not independent workload samples. Because tasks recur across workloads, task-controlled analyses are also necessary even when confidence intervals use workload-level resampling.

The three L3 examples in Figure 3 correspond to `tb2-caffe-cifar-10`, `tb2-train-fasttext`, and `skillsbench-gravitational-wave-detection`. Their labels were checked against the V1 final pool manifest. They illustrate the task pool rather than asserting that all three co-occur in a particular evaluated workload.

C Additional results

Tables 5 and 6 reproduce the remaining These results have not been recomputed from raw trial artifacts. Heavy results are affected by the coupling between agent and verifier timeouts. Missing results are unavailable observations, not zero scores.

Table 5: Additional stream results. Peak is the mean peak in-flight count. Other columns follow Table 2. Timeout-confounded heavy results should be read with caution.

Arm	Tier	n	Reward	Claim	In-flight	Peak	Time (h)
`codex`	heavy	10	0.566	1.000	2.60	5.0	1.29
`glm`	heavy	11	0.611	0.909	5.55	11.5	3.13
`kimi`	heavy	9	0.515	0.939	4.79	11.4	2.98
`cc`	heavy	8	0.502	0.781	6.23	10.9	2.33

C.1 Directory health census

The source report's clean-only rule requires a nonempty reward record, at least 60 steps, no detected output-cap event, no detected throttling, and an available transcript. Table 7 preserves the reported category counts. They sum to 1,440, four fewer than the stated total of 1,444. This discrepancy requires the original census and cannot be resolved by assigning the four directories to a category without evidence.

Table 6: GLM burst means under the earlier full-verification score definition, retained for provenance. These numbers are not comparable with Table 2, which reports mean per-task reward; Section 4 gives the burst comparison under the current definition. Subtracting these aggregate means from the stream table is not a paired estimate.

Condition	Tier	n	Reward	Claim	In-flight	Time (h)
Burst	light	25	0.415	0.984	2.26	1.12
Burst	medium	16	0.600	0.881	3.68	1.66
Burst	heavy	7	0.571	0.957	7.46	1.96

Table 7: Reported directory-health categories. The census is an audit of experiment artifacts, not a population failure-rate estimate.

Category	Directories
Clean	284
Unscored	579
Throttled	302
Short trace (fewer than 60 steps)	198
No transcript	47
Output-cap event	30
Sum of categories	1,440
Stated total in source report	1,444

The threshold is an audit heuristic, not a sufficient definition of valid agent behavior. A short run may represent an interruption, a genuine early exit, or fast completion. Final analysis should classify error events and test sensitivity to the step threshold. Scored runs without transcripts can remain in an outcome audit, but their unrecorded behavior cannot be recovered.

D THE INTAKE-COLLAPSE FAILURE MODE

On heavy workloads Claude Opus 5 retains all of its isolated success rate while GLM-5.3 and DeepSeek-V4-Pro retain 78% and 68% (Table 2). This appendix identifies what the three systems do differently and rules out the experimental explanations. All three run the same harness at the same version, the same reasoning-effort setting, the same disabled tools, the same dispatcher, the same arrival schedule, the same container limits and the same scheduling preamble; the deployed per-trial configurations differ only in the model identifier.

The mechanism is how many tasks a run ever holds at once. Every one of Claude's 15 trials claimed all 20 released tasks, with a peak concurrency of 6 to 19. GLM claimed all 20 in 9 trials and between 5 and 10 in the other six, and in those six its peak concurrency was 1 or 2 — it executed the workload one task at a time. DeepSeek did not claim the full workload in any trial: its peak concurrency is 1 to 3 in 15 of 15, and it claimed 5 to 18 tasks. The resulting never-claimed share is 0%, 25% and 48% of the 300 task instances per system, and median claim latency, the delay from a task becoming claimable to being claimed, is 749, 1187 and 6242 seconds.

The collapsed runs are not idle, short, or faulted. GLM's six collapsed trials are all `clean` under both the operational filter and the looser nearly-clean rule, with zero request-level API errors, zero output-cap events, zero unavailable-route events and zero verifier download failures. Four of the six ran past 3.7 of the 4 available hours. They issue 145 to 281 shell commands and produce 59 to 100 MB of transcript — four to six times Claude's median of 15.5 MB on the same workloads — while calling the task-intake tool only 12 to 17 times. DeepSeek's median transcript is 102 MB against a median of 14 intake calls. The pattern is sustained effort concentrated on a handful of claimed tasks, not a failure to act; two of GLM's collapsed runs also record context-window compaction in their final message, which makes context pressure a candidate trigger without establishing it.

Intake, not task-solving, accounts for the gap. Splitting GLM's trials on whether they claimed the whole workload separates the deficit almost entirely:

GLM-5.3 heavy trials	n	mean reward
claimed all 20 tasks	9	0.624
claimed fewer	6	0.307
Claude Opus 5, all trials	15	0.670

When GLM claims the whole workload it scores within 4.6 points of Claude on the same task set. The 14.8-point aggregate difference is produced by the six collapsed runs, not by weaker per-task work.

For GLM the mode is stochastic; for DeepSeek it is not. The same workload yields both regimes across identical repeats: GLM claims 20, 8 and 8 tasks on one heavy workload and 20, 10 and 5 on another, with byte-identical inputs, arrival schedule and configuration. Single-task-at-a-time execution is therefore a failure mode a run may enter rather than a fixed capability level, and Kimi-K3 shows the same instability on the thirty-task workload, completing 30, 30 and 5 of 30 tasks across three repeats. DeepSeek is the opposite case: 15 of 15 collapsed, which reads as a policy rather than a lapse.

Limits. Six of fifteen is a point estimate over five workloads with three repeats; it does not support a rate. We also do not establish why a run enters the mode — the transcripts record the consequence, and the two compaction events are suggestive, but this collection does not settle the cause. The comparison is between complete agent systems, so harness-side task-intake ergonomics are part of what is being measured and cannot be separated from the model here.

E TASKS WITH A ZERO INDIVIDUAL BASELINE

Table 2 restricts the paired comparison to tasks whose individual baseline is nonzero. This appendix gives the tasks that restriction removes and the evidence that they are construction defects rather than hard tasks.

The exclusion is not specific to one system. Every system has such tasks, and in every system they carry a large positive difference while the remaining tasks carry a negative one. On medium the nonzero-baseline difference runs from -3.1 to -30.2 points across the five systems, while the zero-baseline difference runs from $+5.1$ to $+33.3$; on heavy the two ranges are -3.1 to -28.0 and $+12.5$ to $+45.6$. The zero-baseline tasks number 5 to 7 per system on medium and 11 to 15 on heavy, out of 32 and 53 scored tasks. Because the sign is uniform across systems and strata, the positive contribution is a property of how these tasks were built rather than of any system's scheduling.

Traced causes. Reading the final message of each zero-scoring individual attempt separates the cases. For Claude Opus 5, the system we audited in full, seven of the eighteen zero-baseline tasks end with the agent reporting that the task's own inputs are absent from its container — for example that the OCaml source tree "isn't in this environment", or that "the files it depends on aren't present". The agent stops after ten to twenty steps in these cases. Two more are the renamed-artifact mismatch described below, two end on a request timeout, and one on a provider error; six have no message that identifies a cause.

The missing-input cases share a mechanism with the artifact reuse of Section 3.2, in the opposite direction. A stream shares one filesystem across its tasks, so a file staged for one task is visible to another; individual execution stages only that task's own inputs. Where the individual stage is incomplete, the task is unsolvable in isolation and solvable in the stream, and the paired difference measures the packaging rather than the scheduling.

Renamed artifacts the graders do not follow. The task-assembly step renames per-instance artifacts to avoid collisions when a workload contains several instances of one task. The rename is applied to the container image and to the instruction text but not to the grader. The image stages

`generated-code__<task-id>.hpp` and the instruction requests exactly that file, while the grader reads the unsuffixed `generated-code.hpp`, which is the untouched stub, and reports a mismatch against its reference. The transcripts confirm the agent wrote the file it was instructed to write. Five tasks are affected: two `pareval` sparse-linear-algebra instances and three regex instances. Each scores zero on every individual attempt while its stream counterpart scores one from a sibling's artifact.

What this does and does not resolve. Restricting to nonzero baselines makes the reported differences interpretable, and under that restriction all five systems deliver less under stream execution than in isolation, with no positive value remaining. It does not recover the excluded measurements. The benchmark understates isolated-execution ability on the affected tasks, and any comparison that includes them overstates the benefit of shared execution.

F TERMINAL-STATE ACCOUNTING

Two of the six terminal states in Figure 8 depend on which component writes the record, and the comparison in Appendix J is grouped so that it survives that dependence.

The fixed policies carry high reported-failed shares — 19.7% and 10.7% of released tasks on medium and heavy, against 1.7% and 9.3% for native management — because their driver records a solver session that exits non-zero as a failed task. An agent-managed run has no such driver and may simply stop reporting on a task it still holds, which lands in the claimed-but-unfinished state instead. The two states therefore trade off against each other for reasons of bookkeeping as much as behavior.

Grouping them resolves the ambiguity: greys are work never served and warm shades are work served without credit, regardless of which component wrote the terminal record. That grouping, not the individual state shares, is what the policy comparison rests on. The three reported-complete states are unaffected, since a verifier outcome is written by the same grader in every system, and they sum to the completion column of Table 3 in all eight cells.

G FAILURE-ATTRIBUTION METHOD AND ADJUDICATION

This appendix records how the failure classification of Appendix G is constructed and what the review-class findings contain. The main text reports the resulting distribution; the procedure is here because it governs what those counts are allowed to mean.

Evidence classes. We separate three kinds of evidence and never mix them in one number. *Structured* evidence is stated by an artifact: a harness exit code, a dispatcher terminal state, a container counter. *Derived* evidence is computed from a structured series, such as peak memory against the container's own limit. *Review* evidence is a text match in a transcript, which locates candidates for inspection and is never reported as a rate.

Text matching alone cannot support a rate. Generic out-of-memory phrasing appears 1807 times across 14 of 15 trials in one policy system, while the collection contains one structurally confirmed occurrence. The matched text is predominantly a system narrating an error it is reasoning about ("the build was killed, likely memory, retrying with fewer jobs") rather than a container being terminated. Review evidence is therefore reported as a located case list rather than as a rate.

Resource conflicts. Narrow patterns for four conflict classes match in a minority of trials: disk exhaustion in 56 of 390, process-table or descriptor exhaustion in 47, file-lock contention in 29, and port collisions in 12. Hit counts exceed trial counts by two to four times, which indicates individual transcripts repeating a message rather than many distinct conflicts. A match may be a conflict the system hit, a conflict it recovered from without consequence, or text it reproduced while reasoning, and the records do not distinguish these without inspection. We therefore report a located case list supporting the observation that shared-container execution produces contention, and derive no conflict rate.

Adjudicating references to the harness log tree. We adjudicated by hand every reference to the harness log tree that our classifier placed inside an issued command rather than in tool output:

52 candidates across 70 of 390 trials. No trial references the arrival schedule, which would be the high-severity case. Most candidates are environment reconnaissance: directory listings of the harness log tree, which is empty during the execution phase. Nine are a system rewriting the output paths inside a task's own test script so that its results land in a scratch directory, which redirects grader output away from the harness tree rather than into it. Four are the content of a task's test script appearing in tool output and misclassified as a command, so the in-command count is itself an overestimate. The lower-severity pattern, reading a task's own bundled tests, occurs in 111 trials and is expected: those tests ship with the task.

One candidate is a genuine write. A DeepSeek trial issued `echo 1 > /logs/verifier/reward.txt`, and the harness reward reader does consult that flat path as its last fallback, after a per-task path. The write had no effect on the score: the task is recorded at 0.0 in the final breakdown while the system's own note for it asserts "reward.txt=1" and "3/3 passed", and the per-task reward record contains 0 with a modification time after the execution phase ended, so the authoritative grader ran afresh and contradicted the claim. This is the outcome the fallback ordering predicts, since the authoritative grader always writes the per-task path that takes precedence.

The validity argument therefore rests on the contradicted write, which is empirical, rather than on an argument that the write was impossible: the agent's log-tree writes do reach the host-side trial directory, and one flat grader output file carries a modification time inside the execution window. The reported numbers are unaffected, and the flat fallback path remains reachable by an agent command.

H Details of the experiment setup

All values in this section describe the deployed per-trial configuration and the recorded trial artifacts. Harbor redacts secrets when serializing the agent environment, so a masked value indicates a variable that was set with an unrecoverable value; only an absent key is a defect. We name environment variables without values.

H.1 Agent systems as configured

Within an system, harness version and model identifier are identical across all jobs and tiers. Claude Opus 5, GLM-5.3, and DeepSeek-V4-Pro run on Claude Code 2.1.241; GPT-5.6-sol on Codex 0.50.0; Kimi-K3 on Kimi Code 0.36.1. The Claude Code systems set permission mode to bypass, reasoning effort to `high`, and disable the `WebSearch` and `WebFetch` tools. Codex sets reasoning effort to `high` and uses the `responses` wire protocol.

The systems are not identical apart from the model. The Kimi Code system carries only a version in its harness arguments: no reasoning-effort setting, no permission mode, and *no tool disabling*. Codex likewise sets no tool disabling. Web search and fetch are therefore unavailable to three systems and available to two, which is a configuration difference between the evaluated systems and not a property of the models. Cross-system comparisons should carry this caveat.

Requests route through one gateway host. The Claude Code systems set an Anthropic-compatible base URL and an authentication-token variable; Codex sets an OpenAI-compatible base URL and key variable. The harness-written Codex configuration shows that 50 Codex trials, including 28 of 30 medium-stream and 21 of 27 medium trials, reached the model through a direct route rather than the shared gateway, while the wire protocol stayed `responses` throughout. This is a service-level difference in the collected data.

The output-token cap is set per system through the Claude Code environment. The GLM systems request 131,072 tokens, exactly the gateway ceiling; one gateway-side token more returns an error. The DeepSeek systems request 393,216 tokens, three times that ceiling, so the effective cap is not recoverable from disk and is recorded as unknown. The Opus systems set no cap variable.

H.2 Runtime limits

Three independent limits apply, and which one binds differs between the two settings.

1. **Per-request limit**, `environment.kwargs.request_timeout`: nominally the sandbox API request timeout.
2. **Agent-phase limit**, `agent.override_timeout_sec` and `max_timeout_sec`.
3. **Verifier-phase limit**: null in every configuration in the collection, so a harness default applies.

Table 8: Configured runtime limits by setting and tier. The timeout multiplier is 1.0 and the agent-setup multiplier 5.0 everywhere; the queue wait limit is 3600 s everywhere. The binding limit is marked in bold.

Setting	Tier	Per-request (s)	Agent phase (s)	Container TTL
Stream	Light	**7200**	21600	12 h
Stream	Medium	**9000**	28800	16 h
Stream	Heavy	**14400**	43200	24 h
Stream	Heavy-plus	**21600**	57600	30 h
Individual	All	7200	**5400**	4 h
Stream (10 h serial policy)	Heavy	**36000**	43200	24 h

In stream execution the per-request limit binds and the agent-phase limit is never reached: all 15 heavy serial-FIFO trials end their agent phase between 14,400 and 14,405 seconds, matching the 14,400-second per-request limit, while the agent-phase limit of 43,200 seconds is three times higher and untouched. In individual execution the agent-phase limit binds: the maximum agent-phase duration is exactly 5400 seconds in all four tiers, against a 7200-second per-request limit. The execution budget referred to as 4 hours in Sections 3–4 is therefore the per-request limit, and the 10-hour serial policy raises that same limit from 14,400 to 36,000 seconds, leaving the agent-phase limit unchanged.

The verifier limit appears in no configuration file but is observable: 54 archived trials end their verifier phase at exactly 10,801 seconds (180 minutes), all of them the same energy-market-pricing task variant, across four systems and four archival batches. The limit is adequate rather than tight — one trial completed grading at 7613 seconds and scored full credit, and that is also the maximum verifier duration in the retained set. No retained trial exceeds 10,000 seconds of grading, so the affected runs are archived and contribute to no reported number.

H.3 Measured phase durations

Computed from the four phase timestamp pairs in each trial record, over the 1750 retained trials. Scores are read from the verifier reward record.

Table 9: Phase durations in seconds over all retained trials, and the agent-setup phase by harness.

Phase	Median	p90	Max
Environment setup	201	366	1076
Agent setup	13	75	589
Agent execution	2403	9008	21623
Verifier	33	793	7613
Agent setup by harness			
Claude Code	10	43	120
Kimi Code	14	57	589
Codex	71	110	220

Environment setup costs about 200 seconds per trial and is paid once per container. Individual execution allocates one container per task, so this fixed cost scales with the task count there while stream execution amortizes it across the tasks of a workload. This favors stream execution in the throughput comparison and is one reason those ratios are not efficiency measurements. Codex agent setup is about seven times slower than Claude Code's because it is the only harness that installs Node.js, which is also the origin of the failures in Appendix H.7.

H.4 CONTAINER AND HOST ENVIRONMENT

Each container is provisioned with four CPUs and 8 GiB of memory, confirmed in the container control-group telemetry as a quota of 400,000 μs per 100,000 μs period and a memory limit of 8 GiB. No trial in the collection records any other quota, so resource headroom is held fixed throughout and is not a manipulated variable anywhere in this paper.

The sandbox host reports `x86_64` on a gVisor kernel. One workload image configures its package manager against an `ubuntu-ports` mirror and fetches arm64 package indices; that is a property of the image's package configuration and not of the execution architecture. Conflating the two misdiagnoses the failure in Appendix H.7. Transcripts are returned only when a trial ends, because the sandbox is not mounted during execution, so an absent transcript mid-run is expected and is not evidence of a stalled run.

H.5 TASK PROMPT

The stream workload prompt is a single 2095-byte text, identical by hash across all five systems, all tiers, and the guidance, burst, and resource-labeled conditions. The first trial step carries it for the Claude Code and Kimi Code systems; for Codex the first step is the harness environment context and the prompt is the second step.

```
# Mission
You are a CLI agent running as the natural-language
interface to a sandbox container.  User work arrives over
time as a **stream of tasks** with no fixed count and no
preview of what's coming.  Your job is to **complete as many
tasks as possible, as fast as possible**, inside a single
wall-clock budget.
# Resources
This container is provisioned with:  **4 CPU cores · 8
GB memory**.  Anything you choose to do -- serial work,
parallel work, heavy compute, light I/O -- shares this
single budget.  Treat it as the only hardware constraint
that matters.
```

The allocation is stated literally in the prompt, so any system that changes the container quota must change this text as well or the system will plan against the wrong figure. The guidance condition additionally injects one skill file of 134 lines through a skills directory variable; every other system leaves that variable empty.

Individual execution uses no preamble at all. Its first step is the raw task brief, with no task-pool tools and no instruction about throughput or parallelism. The individual baseline therefore differs from stream execution in prompt as well as in isolation, which limits reading the difference between the two settings as an effect of scheduling alone.

H.6 SAMPLING PLAN

Tier sizes were verified against the arrival declarations rather than assumed: 5, 10, 20, and 30 tasks per workload for light, medium, heavy, and, over 10, 10, 5, and 1 workloads. The last arrival falls between 138 and 358 seconds for light workloads and at 2074 seconds for the single workload. Three attempts are targeted per workload and per individual task throughout. Trial concurrency varies from 5 to 30 across jobs as a throughput setting and is not an experimental variable. Planned trial counts follow: 30 per stream cell at light and medium, 15 at heavy, 3 at, and 69, 42, 93, and 39 per individual cell by tier.

H.7 GPT-5.6-SOL AGENT-SETUP FAILURES

Two gaps in Table 2 belong to the GPT-5.6-sol system: its medium stream cell holds 27 of 30 planned trials, and its individual baseline covers 31 of 32 medium task occurrences. Both trace to a single image-dependent failure in the agent-setup phase, before any model call.

The Codex harness provisions each container with `apt-get install -y ... nodejs npm ...`. On one workload image this command exits 100 with `E: Unable to correct problems, you have held broken packages`. The image's apt configuration points at an `ubuntu-ports` mirror and fetches arm64 package indices, whereas the NodeSource repository serves amd64 packages, and the resulting dependency conflict has no in-image resolution. The three missing stream trials are the three attempts at that one workload, and the missing individual variant is a task drawn from the same image.

Three observations fix this as an infrastructure fault rather than a model or configuration outcome. First, it is deterministic: across 6 live and 23 archived attempts, all 29 trials on this image terminated in setup with no reward record. Second, it is harness-specific: on the same image the four Claude Code and Kimi Code systems completed 3 of 3 attempts each, because only the Codex harness installs Node.js. Third, it is confined to this image; the same system completed every other planned trial in the tier.

We report the affected cells as incomplete rather than substituting a value. The stream cell aggregates the 27 trials that ran, and the coverage gap in the individual baseline and its throughput ratio is recorded in Appendix H. Repairing this requires changing the image's package sources or the harness Node.js installation path, which alters the evaluated configuration and is therefore deferred rather than patched into this evaluation. The container host is x86_64 on a gVisor kernel; the arm64 declaration is a property of the image's package configuration, not of the execution architecture.

I TRAJECTORY EVIDENCE FOR THE INTAKE ACCOUNT

The dispatcher terminal states attribute delivery differences to task intake. The recorded trajectories provide an independent view because the tool calls a system issues show how it spent its turns. the trajectory activity tables in the supplementary tables summarizes activity volume and the tool-composition table its composition.

Activity volume does not predict delivery. GLM-5.3 issues the most activity of any system by a wide margin: a median of 320 steps and 161 model turns per trial against 72 and 37 for Opus, with 15.7M input and 280k output tokens per trial against Opus's 5.4M and 63k. It nonetheless delivers lower SR than Opus in every tier. GPT-5.6-sol occupies the opposite corner, with the fewest steps (median 40) and the fewest tokens (4.3M input), while placing mid-field on SR. Spending more turns is therefore not the mechanism by which a system delivers more of a workload, and step or token counts are not proxies for workload performance.

The tool mix separates the systems along the intake axis. Requests for the next available task account for 11.2% of Opus's tool calls and 16.6% of Kimi's, against 3.1% for GLM and 2.5% for DeepSeek. The ordering matches the never-claimed shares of Section 3 (0.7% and 12.5% against 28.2% and 40.4%) and is visible per trial: on the 30-task workload Opus issues 69 task requests and 20 delegations while DeepSeek issues 3 requests and no delegations, spending 84% of its calls on the shell. GLM and DeepSeek thus direct the large activity volumes into repeated work on tasks they already hold, rather than into acquiring more of the released workload.

Tool-error rates differ by an order of magnitude. The median trial records 2 failed tool results for Opus and 8 for GLM, with 90th percentiles of 6 and 24. Because a failed call consumes a turn, this is one route by which GLM's large turn count yields less delivered work. The Codex trajectories carry no error flag at all, so its column is unavailable rather than zero.

Hidden reasoning is recorded in three incompatible units. Three systems report hidden reasoning and no two report the same quantity. Claude Opus 5 carries per-step thinking-token counts (median 13.9k tokens per trial); GPT-5.6-sol carries a per-trial reasoning-token total (median 5.1k); Kimi-K3 carries a count of reasoning steps (median 34), not convertible to tokens. GLM-5.3 and DeepSeek-V4-Pro run on Claude Code but their gateways return a zero thinking-token field while their trajectories contain reasoning text (66,875 steps for GLM and 24,740 for DeepSeek). A zero or dash in that column indicates what the gateway reported, not how much the system reasoned; no reasoning-effort ordering can be drawn from this data.

Sub-agent delegation is a minority behavior. The median trial in every system spawns no sub-agent. The 90th percentile is 5 for Opus, 6 for Kimi, and 7 for GLM, against 0 for DeepSeek; GPT-5.6-sol is a dash because the Codex harness exposes no delegation tool at all, which differs from DeepSeek's choice not to use one.

J TERMINAL STATES AND RESOURCE UTILIZATION

The policy ordering is carried by never-claimed work. Never-claimed shares fall from 50.0% to 19.7% on medium and from 73.3% to 19.0% on heavy as the policy moves from serial to guidance, while the verified share rises from 33.7% to 50.0% and from 16.3% to 42.7%. Serial leaves 50.0% and 73.3% of released tasks unclaimed and has no unfinished claims, because with $k = 1$ a task is either served to completion or never started. The serial contrast in verified coverage is -0.110 $[-0.190, -0.020]$ on medium and -0.267 $[-0.347, -0.187]$ on heavy (percentile bootstrap over workloads), both excluding zero; fixed $k = 2$ and guidance contrasts do not exclude zero in both tiers.

Holding more tasks does not raise resource use. Figure 9(a) shows that native management on heavy holds nearly four tasks at a time and reaches the same CPU utilization as fixed $k = 2$ holding two (27.2% vs 26.7%); on medium it holds 1.40 tasks at the same 14.6% CPU that serial reaches holding one. Every policy leaves at least 70% of the four provisioned cores idle while tasks wait.

Self-assessment does not improve with management. Of the tasks each policy reported complete, the share the verifier confirmed is 88.6%/79.0% for serial, 87.6%/86.1% for fixed $k = 2$, 82.5%/79.4% for native management, and 79.8%/78.5% for guidance (medium/heavy). The two agent-managed policies are the least accurate in both tiers. Between 11.4% and 21.5% of reported-complete tasks did not verify, so a self-reported completion is not interchangeable with a verified one.

Compute cost per delivered task. CPU-seconds of the agent-execution phase per fully verified task is lowest for GPT-5.6-sol (573–807 CPU-s) and highest for GLM (1255–2145). Across the GLM execution policies, serial FIFO has the *highest* cost per solved task (2172 CPU-s vs 1755 for native management), because its solved count falls faster than its compute does. Serial execution is therefore not a compute-saving configuration.

Memory exhaustion is not a confirmed mechanism. No native-stream trial carries structured evidence of an out-of-memory termination, and exactly one trial elsewhere in the collection does (a burst-arrival run terminated by signal, harness exit code 137). The containers expose no usable out-of-memory counters, so a zero reading is not evidence of absence.

Contention occurs but is not quantifiable; no score was manipulated. Narrow patterns for four resource-conflict classes match in a minority of trials (disk exhaustion in 56 of 390, process or descriptor exhaustion in 47, file-lock contention in 29, port collisions in 12), but a match may equally be a conflict recovered from or text narrated while reasoning. We adjudicated by hand all 52 command-level references to the harness log tree and report zero confirmed instances of score manipulation. One DeepSeek trial wrote a reward file on an agent-writable fallback path; the authoritative grader reran and overrode it, recording 0.0. That fallback should be removed before a released evaluation (Appendix G).

K RELATED WORK

Computer-use agent evaluation. Computer-use benchmarks evaluate agents by the state changes they produce in executable environments. SWE-bench packages software repositories with executable test suites (Jimenez et al., 2024). Terminal-Bench and TUA-Bench package each task with a container specification and an in-environment verifier (Merrill et al., 2026; Chen et al., 2026). CORE-Bench evaluates the reproduction of computational research from code and data (Siegel et al., 2024), while TerminalWorld reconstructs Docker environments and tests from recorded terminal workflows (Chu et al., 2026). On graphical interfaces, WebArena, VisualWebArena, and OSWorld evaluate browser and desktop agents (Zhou et al., 2024; Koh et al., 2024; Xie et al., 2024). Their common limitation

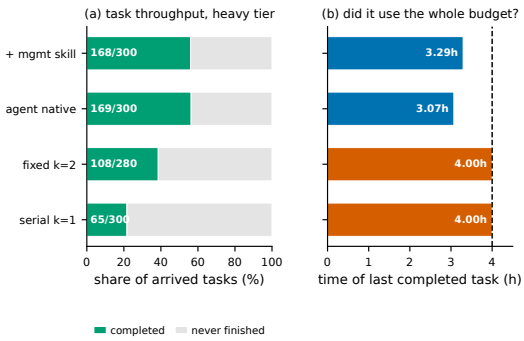

Figure 14: Budget utilization on heavy workloads. (a) Share of released tasks completed; (b) time of last completed task vs. the 4-hour budget. Serial FIFO reaches the budget in all 15 trials; native management stops earlier because claimable work runs out.

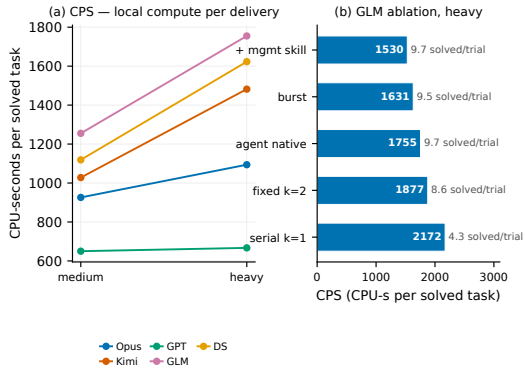

Figure 15: CPU-seconds per fully solved task (setup, grading, and remote inference excluded). (a) By system and tier under stream execution; (b) GLM-5.3 heavy-tier policies. Serial FIFO has the highest cost per solved task despite consuming the least total execution time.

for our setting is the evaluation unit: they score one objective in one environment instance, whereas AOS-Bench composes multiple independently verifiable objectives in a shared container released over time.

Agents for management and sustained operation. A growing line of work evaluates agents as operators of persistent social or economic processes. Vending-Bench, Business Arena, and E-Commerce Bench ask agents to manage inventories, capital, suppliers, and long-horizon economic outcomes (Backlund & Petersson, 2025; Pan et al., 2026; Fan et al., 2026). TheAgentCompany embeds agents in a simulated software company requiring communication, code, and professional tasks (Xu et al., 2025). These environments expose important management capabilities, but their managed object is a domain process; AOS-Bench instead studies the role in which the managed object is the user's computational workload under shared resource constraints.

Agent runtimes and resource management. AIOS factors scheduling, context and memory management, storage, and access control into an agent-oriented kernel (Mei et al., 2025). AgentCgroup characterizes bursty OS-resource use in sandboxed coding agents and applies tool-call-granularity cgroups (Zheng et al., 2026). HiveMind coordinates agents sharing rate-limited endpoints through admission control and priority queues (Agyemang et al., 2026). SpecBox and PASTE reduce sandbox cold-start and reservation costs through speculative prewarming and joint tool-model scheduling (Zhang et al., 2026; Sui et al., 2026). These systems improve how a known set of executions consumes particular resources; AOS-Bench evaluates the higher-level policy that turns arriving user requests into those executions.