AgentCgroup: Understanding and Controlling OS Resources of AI Agents

Yusheng Zheng
UC Santa Cruz
yzhen165@ucsc.edu

Jiakun Fan
Virginia Tech
jiakunfan@vt.edu

Quanzhi Fu
Virginia Tech
quanzhif@vt.edu

Yiwei Yang
UC Santa Cruz
yyang363@ucsc.edu

Wei Zhang
UConn
wei.zhang@uconn.edu

Andi Quinn
UC Santa Cruz
aquinn1@ucsc.edu

Abstract

AI agents are increasingly deployed in multi-tenant cloud environments, where they execute diverse tool calls within sandboxed containers, each call with distinct resource demands and rapid fluctuations. We present a systematic characterization of OS-level resource dynamics in sandboxed AI coding agents, analyzing 144 software engineering tasks from the SWE-rebench benchmark across two LLM models. Our measurements reveal that (1) OS-level execution (tool calls and container/agent initialization) accounts for 56–74% of end-to-end task latency; (2) memory, not CPU, is the concurrency bottleneck; (3) memory spikes are tool-call-driven with a 15.4× peak-to-average ratio; and (4) resource demands are highly unpredictable across tasks, runs, and models. Comparing these characteristics against serverless, microservice, and batch workloads, we identify three mismatches in existing resource controls: a *granularity mismatch* (container-level policies vs. tool-call-level dynamics), a *responsiveness mismatch* (user-space reaction vs. sub-second unpredictable bursts), and an *adaptability mismatch* (history-based prediction vs. non-deterministic stateful execution). We propose AgentCgroup, an eBPF-based resource controller that addresses these mismatches through hierarchical cgroup structures aligned with tool-call boundaries, in-kernel enforcement via sched_ext and memcg_bpf_ops, and runtime-adaptive policies driven by in-kernel monitoring. Preliminary evaluation demonstrates improved multi-tenant isolation and reduced resource waste.

1 Introduction

AI coding agents such as Claude Code [1], OpenHands [34], and SWE-agent [37] combine large language models with autonomous tool use through a reason-then-act loop [38] to address tasks such as software engineering [33] and OS turing [40]. These agents execute tool calls (compilers, test runners, package managers) inside sandboxed containers [2, 11]. With major vendors like OpenAI [28], GitHub [12], Google [13], and Devin [8] now offering such capabilities as commercial products and cloud providers hosting many concurrent agent instances on shared infrastructure [6, 17, 21], efficient resource management becomes critical.

Effective resource management requires understanding workload behavior, yet a systematic characterization of the OS-level resource dynamics of these agent workloads is lacking. To fill this gap, we characterize these dynamics using Claude Code [1] across 144 SWE-rebench [5, 16] tasks with two LLM backends: Claude Haiku 4.5 (cloud API) and GLM-4.7-Flash (local GPU). Our analysis reveals four key findings: (1) OS-level execution (tool calls and container/agent initialization) accounts for 56–74% of end-to-end task latency, with LLM reasoning accounting for 26–44%; (2) memory, not CPU, is the primary bottleneck for multi-tenant concurrency density; (3) memory exhibits a two-layer structure with a stable ~185 MB framework baseline plus tool-call-driven bursts reaching a 15.4× peak-to-average ratio; and (4) resource demands are highly unpredictable, varying 20× across tasks and 1.8× across runs of the same task.

Comparing these characteristics against existing resource controls reveals three mismatches: a *granularity mismatch*, where container-level policies waste most allocated memory or trigger OOM kills; a *responsiveness mismatch*, with user-space reactions at millisecond-to-minute timescales too slow for sub-second unpredictable bursts; and an *adaptability mismatch*, as history-based prediction is poorly suited to non-deterministic execution where kill-and-restart destroys accumulated LLM context.

We propose AgentCgroup, an eBPF-based resource controller that addresses these mismatches through hierarchical cgroup v2 structures aligned with tool-call boundaries, in-kernel enforcement via sched_ext [15] and memcg_bpf_ops [41] for microsecond-level reaction, and runtime-adaptive policies driven by in-kernel monitoring, with graceful degradation.

Our contributions are:

- **Characterization.** A systematic measurement of OS-level resource dynamics in sandboxed AI coding agents (144 tasks, 2 models), yielding the four findings above (§3).
- **Mismatch analysis.** A quantitative comparison with serverless, microservice, and batch workloads, identifying the three mismatches above (§4).
- **System.** AgentCgroup, with preliminary evaluation on real agent traces showing 29% lower high-priority P95 latency under multi-tenant memory contention (§5, §6).

2 Background

AI Coding Agents. Modern AI coding agents combine large language models (LLMs) with tool-use capabilities to autonomously solve software engineering tasks. These agents operate in an iterative loop: the LLM reasons about the current state and emits a structured tool-use request (specifying the tool name and arguments); the agent framework parses this request, forks a subprocess inside the sandboxed container to execute the tool (e.g., running `pytest`, editing a file, invoking a compiler), collects the result, and returns it to the LLM for the next reasoning step. Representative

systems include both open-source frameworks (OpenHands [34], SWE-agent [37]) and commercial products (Claude Code [1], Cursor [10], Apple Xcode agentic coding [3]). Each tool invocation spawns distinct processes with varying resource profiles; for example, a compiler may consume gigabytes of memory, while a simple file read uses minimal resources. This creates highly dynamic, phase-varying workloads that challenge traditional resource management approaches [4, 7, 21, 22, 27]. Recent work has begun analyzing agent costs from an inference infrastructure perspective [17, 39] and proposing OS-level abstractions for LLM agents [6, 25], but to our knowledge none provides a systematic characterization of the OS-level resource dynamics inside agent sandboxes or designs kernel-level controls informed by such measurements.

Linux cgroup provides a hierarchical resource governance abstraction where the kernel organizes tasks into a tree of control groups and applies controller-specific accounting and enforcement along that hierarchy [14]. The memory controller exposes two key boundaries: `memory.high` as a soft throttle point that triggers reclaim pressure without invoking the OOM killer, and `memory.max` as a hard limit that triggers OOM when exceeded. Cgroup v2 also provides lifecycle controls: `cgroup.freeze` stops all processes in a subtree until unfrozen, `cgroup.kill` terminates all processes while handling concurrent forks, and `memory.oom.group` ensures atomic OOM termination to avoid partial failures.

eBPF enables Linux to address the tension between standardized interfaces and dynamic workloads by introducing programmable enforcement points, providing a safe and dynamically loadable mechanism for executing control logic inside the kernel [23]. On the CPU side, `sched_ext` allows scheduling policies to be defined by BPF programs with fail-safe reversion to default behavior on errors [15]. On the memory side, `memcg_bpf_ops` introduces hooks such as `get_high_delay_ms` for custom throttle delays on `memory.high` breaches [41]. These primitives enable in-kernel enforcement with microsecond-level reaction times.

3 Agent Workload Characterization

Designing effective resource controls requires understanding workload behavior, as prior studies of serverless [30], microservice [9], and batch [32] workloads have shown. We characterize AI coding agent workloads along three axes, each bearing on a different aspect of resource control: the execution model, which determines the granularity at which resources vary (§3.2); temporal resource dynamics, which determine how fast controls must react (§3.3); and cross-task variance, which determines whether demands can be predicted (§3.4).

3.1 Experimental Setup

All experiments run on a single machine (Intel Core Ultra 9 285K, 24 cores, 128 GB DDR5, Ubuntu 24.04, Linux 6.15.11, cgroup v2). Each task executes in an isolated Podman container using official SWE-rebench [5] images (2.9–17.3 GB each) with no resource limits applied. The agent framework (Claude Code) and its dependencies are bind-mounted from the host into each container, eliminating per-container installation overhead. Initialization overhead thus comprises two phases: Podman's user-namespace ID remapping of the image's overlay layers (dominant, scaling with image size), followed by agent framework startup (Node.js process and API connection). Image pull time is excluded from all reported initialization measurements. We use two underlying models: (1) Claude Haiku 4.5 (cloud API) and (2) GLM-4.7-Flash (local GPU). We collect 111 tasks with GLM and 33 tasks with Haiku from SWE-rebench [5]; the 33 Haiku tasks are a subset of the 111 GLM tasks, so all cross-model comparisons use this shared overlap. For each task, we sample CPU utilization and memory usage at 1-second intervals and record each tool call's type and timestamps. Tool execution time is measured from when the LLM emits a tool-use request to when the corresponding result is returned, encompassing framework dispatch, process execution, and result collection.

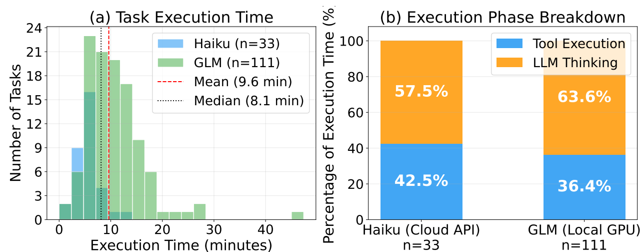

Figure 1: Task execution time distribution (a) and execution phase division (b).

3.2 Execution Model

Each agent task runs for 5–11 minutes (GLM mean 10.8, Haiku mean 5.8, overall median 8.1; Fig. 1(a)), far longer than serverless invocations (100 ms–2 s) yet shorter than batch jobs, and executes stateful multi-round reasoning and tool-call loops within a sandboxed container.

LLM reasoning accounts for 26–44% of end-to-end task latency; the remainder is consumed by tool execution (~40% of active time) and initialization (29–45%). As shown in Fig. 1(b), excluding initialization overhead, tool execution accounts for a mean of 42.5% (Haiku) and 36.4% (GLM) of active time (median 34.7% and 36.5% respectively), though individual tasks range from 0% to 86% (Fig. 3(a)). Over the full task lifecycle, container and agent initialization accounts for 31–48%, tool execution ~26%, and LLM reasoning 26–44%. OS-level overhead (initialization + tool execution) thus accounts for 56–74% of user-perceived task completion time.

Bash dominates tool execution, spanning three orders of magnitude in duration. In Haiku, Bash and the sub-agent tool (which delegates to a new agent instance) together account for over 90% of total tool execution time (47.8% and 43.2% respectively); GLM relies almost entirely on Bash (98.1% of tool time, Fig. 2(a)). Execution times span three orders of magnitude: sub-agent calls average ~100 s, Bash commands 4–6 s, and lightweight tools such as Read and Edit under 0.5 s. The two models adopt different strategies: Haiku distributes work via sub-agent calls and Web search, while GLM concentrates all computation in local Bash calls.

Further analyzing Bash command semantics (Fig. 2(b)), test execution (pytest, unittest, etc.) dominates Bash time in both models (Haiku 72.9%, GLM 43.7%), followed by package installation (~10%)

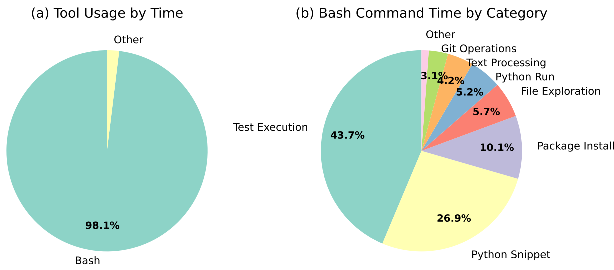

Figure 2: Tool execution time distribution (a) and Bash command semantic category proportion (b), GLM agent.

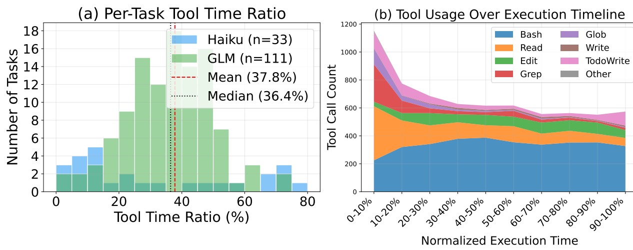

Figure 3: Tool time proportion distribution (a) and tool call distribution over execution progress (b), all 144 tasks.

and Python snippets (GLM 26.9%). Resource demands vary accordingly: test execution is CPU- and memory-intensive, while file exploration and Git operations are lightweight. Tool calls also follow a temporal "understand-modify-verify" pattern (Fig. 3(b)): Read concentrates in the first 30% of execution, Bash peaks in 40–80%, and Edit is distributed evenly.

3.3 Resource Dynamics

We now examine whether agent resource profiles match the moderate, predictable patterns assumed by existing controllers.

Memory, not CPU, is the primary bottleneck for multi-tenant concurrency density. Agent average CPU utilization is low (Haiku 13.2%, GLM 7.6%, normalized to one core, i.e., 100% = one fully utilized core), well below saturation on the 24-core experimental platform. Peak memory can reach 2–4 GB, so 128 GB of RAM with peak allocation supports only 32–64 instances, while CPU utilization at this concurrency remains below 36% of total capacity.

Resource consumption exhibits a two-layer structure: a ∼185 MB framework baseline plus tool-call bursts. As shown in Fig. 4(b), the agent framework (e.g., Claude Code's Node.js runtime) maintains a stable memory baseline: across all 144 tasks, early-execution memory averages about 185 MB (Haiku 183, GLM 188). Resource fluctuations come almost entirely from subprocesses spawned by tool calls: test execution, dependency installation, and other operations raise memory to 500 MB–2 GB, then fall back to baseline. Aggregated memory traces (Fig. 4(b)) confirm a stable first half (∼185–200 MB) with increasing variance in the second half as Bash-dense phases intensify. In multi-tenant scenarios, 64 concurrent instances require about 12 GB of memory for framework baseline alone, on top of which tool bursts add an order-of-magnitude higher peak demand.

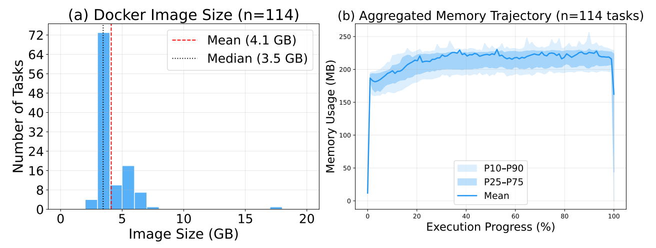

Figure 4: Docker image size distribution (a) and aggregated memory trajectory (b), all 144 tasks.

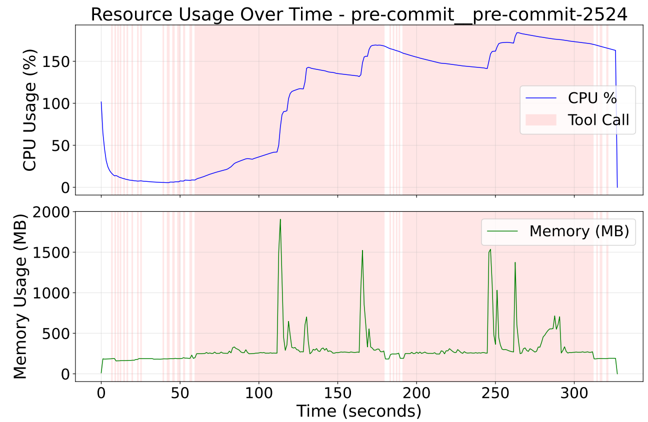

Figure 5: Resource usage time series: Haiku agent executing pre-commit/pre-commit#2524.

Within the burst layer, resource consumption is determined by what the tool *does* (e.g., pytest vs. git status), not which tool is invoked (e.g., Bash vs. Read): Bash calls differ by 13.7× in peak memory depending on the command executed. For example, Bash calls in pydicom/pydicom#2022 (a medical imaging library) consume average peak memory of 4 GB, while streamlink/streamlink#2160 (a network streaming tool) needs only 291 MB. Across Bash command categories, test execution (pytest, etc.) P95 memory spike reaches 518 MB (Haiku)/234 MB (GLM) with average CPU spike +3.2%; package installation P95 spike is about 233 MB; file exploration and Git operations average only 4.5 MB and 13.5 MB respectively.

Resource usage follows a burst-silence pattern, with 98.5% of memory bursts occurring during tool calls. Agent workloads exhibit large temporal fluctuations: memory changes by up to 2.9 GB within a single 1-second interval and CPU peaks exceed 100% (multi-core). As shown in Fig. 5 and Fig. 6, resource spikes align closely with tool calls, while LLM reasoning phases show stable, low usage. Classifying each 1-second sample by phase and counting memory bursts exceeding 300 MB (∼1.6× the framework baseline): in Haiku, tool calls occupy only 50.6% of sampling time yet contain 98.5% of memory bursts; in GLM, 35.9% of time contains 67.3% of bursts (1.9× concentration). CPU burst attribution is more dispersed (Haiku 55.3%, GLM 30.2%), because GLM's local inference generates steady CPU load even outside tool calls.

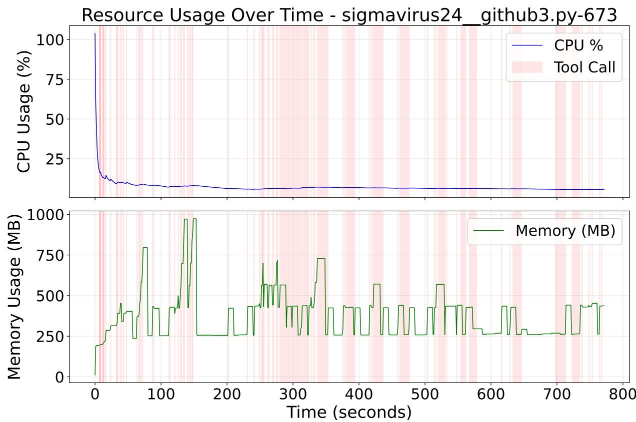

Figure 6: Resource usage time series: GLM agent executing sigmavirus24/github3.py#673.

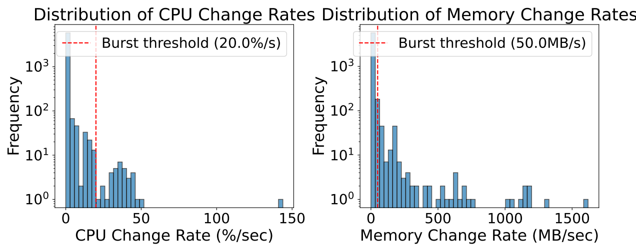

Figure 7: Resource change rate distribution (CPU and memory), Haiku dataset.

Resource bursts last 1–2 seconds with peak-to-average ratio up to 15.4×, several times beyond traditional cloud workloads. The most extreme case, pydicom/pydicom#2022, has peak memory of 4060 MB while average memory is only 264 MB, a 15.4× ratio; this peak falls back to the 230 MB baseline within seconds (Fig. 5, Fig. 6). Memory peaks concentrate in the latter half of execution (mean/median ~65% progress) but occur throughout the execution cycle. Change rates are also large: maximum memory change rate reaches 3 GB/s, CPU change rate exceeds 50%/second, with 1.7%–3.8% of 1-second intervals showing memory changes exceeding 100 MB. Defining burst thresholds at 20%/s for CPU and 50 MB/s for memory (dashed lines in Fig. 7), a substantial tail of sampling intervals exceeds these thresholds.

85%–97% of tasks contain retry loops with progressive memory accumulation. Retry is common in agent workloads: 85% (28/33) of tasks in the Haiku dataset contain retry groups (three or more consecutive Bash calls executing the same test command, e.g., repeated `pytest` invocations); in the GLM dataset, this ratio reaches 97% (108/111). GLM averages 3.9 retry groups per task, with up to 56 consecutive retries, consuming 7–21% of total execution time (Haiku 7.4%, GLM 20.5%). Each retry cycle retains previous memory context without cleanup, causing progressive memory accumulation (up to 502 MB unreleased in the worst case).

CPU-memory correlation varies by task (−0.84 to +0.50); co-directional change cannot be assumed. The average correlation coefficient across all tasks is −0.39: some tasks show positive correlation (tool execution pulls up both CPU and memory simultaneously), while others show negative correlation (CPU-intensive phases coincide with lower memory demand).

Agent container images average 3.5 GB, 7× larger than typical microservice images and 70× larger than serverless functions. As shown in Fig. 4(a), image sizes concentrate in 3–4 GB (range 2.9–17.3 GB across 114 deduplicated images, totaling 456 GB for 111 GLM tasks).

3.4 Unpredictability

Evidence suggests high non-determinism: 1.8× execution time variance across runs of the same task. We executed the same task (iterative/dvc#777) three times, observing execution times of 402, 222, and 259 seconds respectively, a 1.8× difference. The three runs produced entirely different solutions: different code modifications, file counts, and strategies. This non-determinism stems from LLM reasoning randomness and decision-path diversity. Even LLM-observable proxies are uninformative: conversation rounds correlate moderately with execution time (r = +0.57 to +0.82) but not with peak memory ($r < 0.11$), confirming that resource consumption is driven by what tools execute (e.g., pytest vs. file read) rather than reasoning scale.

Resource demands vary 20× across tasks and diverge further across models. Across our dataset, peak memory requirements range from 197 MB to 4 GB (CV = 147%): scientific computing tasks (numba/numba#5721, pydicom/pydicom#2022) exhibit significantly higher memory than CLI tools (joke2k/faker#1520) or network utilities (streamlink/streamlink#2160), yet all run in the same container. Model choice amplifies this variation: Haiku and GLM show 1.7× CPU utilization difference on the same tasks (Haiku mean 13.2% vs. GLM 7.6%). Haiku's cloud API inference consumes more local CPU for response parsing and context management; GLM's local GPU inference shifts CPU load almost entirely to tool calls (only 0.5% of sampling points exceed 50% CPU, vs. Haiku 8.2%).

4 Resource Management Mismatches

The characterization in §3 reveals that agent workloads differ from known cloud workloads along several dimensions (Table 1). These differences create three resource management mismatches (Table 2): granularity, responsiveness, and adaptability. We analyze each mismatch below, explaining why solutions from kernel cgroup interfaces to cluster-level autoscaling do not adequately address them.

4.1 Granularity Mismatch

Agent resource demands vary at tool-call granularity, but all existing controls set a single policy at container level. The memory peak-to-average ratio illustrates this most clearly: Azure Functions exhibit near-flat memory [30], Azure VMs stay within 2–3× [9], and Google Autopilot recommends within 2× of actual peaks [29], whereas agent workloads exhibit far higher ratios (§3).

The mismatch has two dimensions: temporally, container-level policies cannot track tool-call-level dynamics; across resource types, CPU and memory are managed jointly despite being decoupled in agent workloads. On the temporal axis, cgroup v2 hard limits (`memory.max`) force a binary choice: setting to peak (4060 MB in the worst case) wastes 93% of allocated memory (needed only 2% of the

Dimension	Serverless/FaaS [30]	Microservices [9]	Batch/HPC [32]	AI Coding Agent
Execution duration	100ms–2s	Long-running	Minutes–hours	**5–11 minutes**
Container image	~50 MB	100 MB–1 GB	1–10 GB	**2.9–17.3 GB (med. 3.5)**
Statefulness	Stateless	External state	Stateful	**In-process stateful**
Memory footprint	128–512 MB	Steady ~1 GB	Scales with data	**185 MB idle, peaks 2–4 GB**
Memory peak/avg	~1.5×	2–3×	~1×	**15.4×**
CPU utilization	Brief spike	10–40%	80–100%	**<13% avg, peaks >175%**
Determinism	Deterministic	Mostly deterministic	Deterministic	**1.8× variance for same task**
Resource pattern	Flat	Steady + daily cycle	Stable rise	**Burst-silence alternating**
Termination cost	Just retry	Can migrate	Lose progress	**Lose all LLM context**

Table 1: Quantitative comparison of AI coding agent workloads with typical cloud workloads.

	Static Limits	Reactive Control	Predictive Scaling
Tools	mem.max/high, cpu.max [14]; K8s QoS [19]	PSI [35]; oomd [26, 31]; TMO [36]	VPA [20]; Autopilot [29]
Assumes	Known peak; stable demand	Gradual pressure; kill acceptable	Repeatable; history valid
Agent	15.4× peak/avg; tool-semantic variation	1–2 s burst; unpredictable timing	1.8× variance; kill = lose context
Mismatch	Granularity	Responsiveness	Adaptability

Table 2: Existing resource management tools and their mismatches with agent workloads.

time), while setting to average triggers OOM kills during 1–2 second tool bursts, destroying all accumulated agent state. Retry loops (§3) compound this, as progressive memory accumulation means a limit adequate for early iterations may trigger OOM by the fifth. Soft limits (`memory.high`) fare no better: reclaim cannot distinguish the ~185 MB framework baseline (Node.js heap, V8 JIT cache) from tool subprocess memory, causing GC pressure that degrades LLM response parsing; moreover, a single container-level threshold cannot differentiate `git status` (13.5 MB) from `pytest` (518 MB P95). Kubernetes QoS classes [19] face both limitations: Guaranteed incurs comparable waste, BestEffort risks stateful agent termination, and Burstable still cannot set per-tool-call quotas; they also tie CPU and memory into a single class, despite memory bursts concentrated in tool calls (98.5%) while CPU bursts are more dispersed (55.3%), and CPU–memory correlation varying from −0.84 to +0.50 across tasks (§3). The large image footprint further precludes spawning additional containers for finer granularity (initialization consuming 31–48% of task time, §3), reinforcing the need for sub-container resource domains.

4.2 Responsiveness Mismatch

Agent resource bursts last only 1–2 seconds with change rates reaching several GB/s, a burst-silence pattern distinct from all traditional workload categories (Table 1). Moreover, their timing is unpredictable due to non-deterministic tool-call sequences. This combination of fast and unpredictable bursts means controllers need to react in real time at kernel speed. PSI-driven solutions (systemd-oomd [31], Meta oomd [26]) monitor cgroup Pressure Stall Information and take action (kill the cgroup or trigger swap) when pressure exceeds a threshold. Their design assumes (a) a sufficient time window for decision-making after the pressure signal appears, and (b) that kill or migration is an acceptable degradation path. Both assumptions fail for agents: the full cycle from PSI signal generation to user-space daemon reception, decision, and cgroup control file write takes tens of milliseconds, by which time the burst has already passed or triggered kernel intervention. PSI is also a container-level aggregate that cannot attribute pressure to specific tool calls; it reports "container has pressure" but not "pytest caused it." Kubernetes VPA [20] adjusts resources at Pod restart level (stable) or minute-level in-place resize [18] (alpha feature), orders of magnitude slower than second-scale tool bursts, making within-execution adjustment impossible. When slow reaction does lead to OOM, recovery is itself slow: restarting a multi-GB agent container incurs initialization overhead consuming 31–48% of typical task time (§3), amplifying the cost of every missed burst.

4.3 Adaptability Mismatch

Traditional cloud workloads are largely deterministic, allowing history-based resource management: Borg's [32] utilization data guides allocation, and Autopilot [29] uses historical P95 metrics for automatic recommendations. Agent workloads violate this assumption along three axes. First, resource demand varies over an order of magnitude across tasks (§3), so values recommended by VPA [20] are inevitably too high or too low. Second, even the same task produces different resource demands across runs (1.8× execution time variance), and output token count has near-zero correlation with peak memory ($r = -0.14$), making historical percentiles statistically meaningless. Third, within a single execution, retry loops (§3) cause progressive memory accumulation, so memory demands grow unpredictably even during one run, not only across runs.

This unpredictability is compounded by the high cost of the traditional fallback: kill-and-restart imposes a *triple penalty* on agent workloads. First, *slow recovery*: multi-GB agent container images make cold-start recovery (pulling layers, re-initializing the container and agent framework) consume 31–48% of total task time (§3), orders of magnitude longer than serverless cold starts. Second, *lost state*: an OOM kill destroys minutes of accumulated in-process LLM context, and unlike microservices with external state stores, this context cannot be checkpointed or migrated. Third, *non-deterministic re-execution*: re-running the same task follows an entirely different solution path (§3), so restart does not even guarantee convergence to the original solution. Any strategy that

relies on termination as fallback, including Kubernetes eviction, systemd-oomd [31], and static OOM kill, incurs all three penalties simultaneously. Effective resource management should therefore adapt at runtime based on real-time observation, with graceful degradation (throttling, freezing) rather than termination.

5 AgentCgroup Design and Implementation

AgentCgroup addresses the three mismatches identified in §4: fine-grained resource domains match tool-call-level dynamics; in-kernel eBPF enforcement reacts at microsecond timescales; and runtime-adaptive policies replace historical prediction with in-kernel monitoring and graceful degradation.

Fine-grained resource domains. The granularity mismatch (§4) arises because existing controls set a single policy per container, while agent demands vary per tool call. AgentCgroup organizes resources using a hierarchical cgroup v2 structure where each agent workload maps to a cgroup node with tool calls as child nodes, enabling per-tool-call resource constraints while maintaining overall workload budgets. For recovery, AgentCgroup uses cgroup v2 lifecycle primitives: freezing subtrees when tool calls exceed soft limits, and atomically killing subtrees when termination is necessary.

In-kernel enforcement. The responsiveness mismatch (§4) arises because user-space controllers react at millisecond-to-minute timescales, while agent bursts last 1–2 seconds with unpredictable timing. AgentCgroup executes control logic directly at kernel cgroup enforcement points via eBPF at the per-tool-call child cgroups described above, enabling microsecond-level reaction without user-kernel round trips. On CPU, AgentCgroup uses `sched_ext` [15] maintains per-workload and per-tool-call metadata in BPF maps, prioritizing latency-sensitive tool calls with automatic fail-safe reversion on errors. On memory, AgentCgroup uses `memcg_bpf_ops` hooks [41] to implement custom throttling delays when a tool call cgroup breaches its soft limit or hard limit. When memory pressure rises, AgentCgroup eBPF program applies graduated responses based on priority (throttling via `memory.high` delays, freezing via `cgroup.freeze`) rather than termination, preserving agent state.

Runtime-adaptive policies with graceful degradation. The adaptability mismatch (§4) arises because history-based prediction fails for non-deterministic agent workloads, and kill-restart destroys accumulated LLM context. AgentCgroup uses eBPF to trace process creation and memory allocation changes in-kernel, detecting tool-call boundaries and resource dynamics in real time to determine per tool call control behavior without user-space polling. A lightweight user-space daemon handles cgroup lifecycle management and policy configuration via shared BPF maps. We implement a proof-of-concept prototype of AgentCgroup in C using libbpf and BPF CO-RE [24]. The prototype extends Agentsight[39] monitor and implements in-kernel memory enforcement via memcg_bpf_ops [41] and CPU scheduling via sched_ext [15], and runs on Linux 6.15 with memcg_bpf_ops patches (currently under upstream review).

6 Preliminary Evaluation

We evaluate AgentCgroup by replaying real agent memory traces from §3 at 50× accelerated speed in a multi-tenant setting with AgentCgroup enforcement. Experiments run on an Intel Core Ultra 7 258V (4 cores, 16 GB RAM) with a patched Linux 6.19.0-rc5 kernel (bpf-next tree plus the memcg struct_ops RFC patches [41]). No application code is modified; isolation is achieved entirely through container cgroup boundaries and eBPF hooks. We use three agent traces run concurrently in separate cgroups: dask/dask#11628 as the HIGH-priority session (peak 421 MB, `memory.high = max`) and two instances of sigmavirus24/github3.py#673 as LOW-priority sessions (peak 406 MB each, `memory.high = 400 MB`). Under BPF, LOW cgroups are throttled when the HIGH cgroup experiences memory pressure; the HIGH cgroup is protected via `below_low`. We compare against a no-isolation baseline under two memory pressure scenarios (Fig. 8).

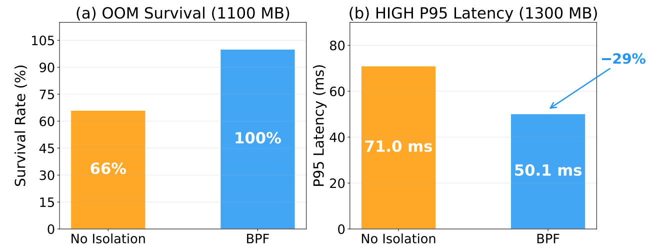

Figure 8: BPF enforcement evaluation with real agent trace replay. (a) OOM survival rate under tight memory (1100 MB total for ~1233 MB demand). (b) HIGH-priority P95 allocation latency under moderate memory (1300 MB).

Results. Under tight memory (1100 MB total for ~1233 MB combined demand), the baseline OOM-kills one LOW process (66% survival); BPF allows all processes to complete (100%) by throttling LOW allocations (239 delay triggers) while HIGH finishes with only +2.8% overhead (Fig. 8(a)). BPF also reduces HIGH-priority P95 allocation latency by 29% (70.97→50.14 ms) through reduced memory contention (Fig. 8(b)). Enforcement overhead is negligible: P50 latency increases by 0.3% and total completion time decreases by 1.1%. Separately, kernel selftests with a 2000 ms configured delay confirm BPF throttling precision within 2.3% relative error (2.000 ± 0.046 s measured).

7 Conclusion and Future Work

We characterized AI coding agent tasks across two models, finding OS execution accounts for 56–74% of latency with a memory peak-to-average ratio of 15.4×. Based on these findings, we presented AgentCgroup, an eBPF-based resource controller addressing granularity, responsiveness, and adaptability mismatches through tool-call-aligned cgroup domains in memory and CPU, microsecond-level reaction, and runtime-adaptive policies. Our current evaluation is limited to trace replay with a proof-of-concept prototype; the characterization covers one agent framework and one benchmark. Future work will evaluate AgentCgroup with live agent workloads at production scale, extend characterization to diverse agent frameworks and domains beyond coding, and explore fine-grained resource control across diverse container runtimes. Moreover, the current prototype mainly focuses on cpu and memory resource control; challenges such as initialization overhead, large container images, and retry-induced accumulation remain open and require optimizations across the sandbox runtime and orchestration stack.

References

[1] Anthropic. 2026. How Claude Code Works. Claude Code Documentation. https://code.claude.com/docs/en/how-claude-code-works (accessed 2026-02-09).

[2] Anthropic. 2026. Securely Deploying AI Agents. Claude API Documentation (Agent SDK Guides). https://platform.claude.com/docs/en/agent-sdk/secure-deployment (accessed 2026-02-09).

[3] Apple. 2026. Xcode 26.3 unlocks the power of agentic coding. Apple Newsroom. https://www.apple.com/newsroom/2026/02/xcode-26-point-3-unlocks-the-power-of-agentic-coding/ (accessed 2026-02-09).

[4] Zain Asgar, Michelle Nguyen, and Sachin Katti. 2025. Efficient and Scalable Agentic AI with Heterogeneous Systems. arXiv preprint arXiv:2507.19635.

[5] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. arXiv preprint arXiv:2505.20411. Dataset: https://huggingface.co/datasets/nebius/SWE-rebench.

[6] Teofil Bodea, Masanori Misono, Julian Pritzi, Patrick Sabanic, Thore Sommer, Harshavardhan Unnibhavi, David Schall, Nuno Santos, Dimitrios Stavrakakis, and Pramod Bhatotia. 2025. Trusted AI Agents in the Cloud. arXiv preprint arXiv:2512.05951.

[7] Jinyuan Chen, Jiuchen Shi, Quan Chen, and Minyi Guo. 2025. Kairos: Low-latency Multi-Agent Serving with Shared LLMs and Excessive Loads in the Public Cloud. arXiv preprint arXiv:2508.06948.

[8] Cognition. 2024. Introducing Devin, the first AI software engineer. Cognition Blog. https://cognition.ai/blog/introducing-devin (accessed 2026-02-09).

[9] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. ACM, 153–167.

[10] Cursor Team. 2026. Best practices for coding with agents. Cursor Blog. https://cursor.com/blog/agent-best-practices (accessed 2026-02-09).

[11] David Dworken and Oliver Weller-Davies. 2025. Beyond Permission Prompts: Making Claude Code More Secure and Autonomous. Anthropic Engineering Blog. Published Oct 20, 2025. https://www.anthropic.com/engineering/claude-code-sandboxing (accessed 2026-02-09).

[12] GitHub. 2026. About GitHub Copilot coding agent. GitHub Documentation. https://docs.github.com/en/copilot/concepts/agents/coding-agent/about-coding-agent (accessed 2026-02-09).

[13] Google. 2025. Jules, Google's asynchronous AI coding agent, is out of public beta. Google Blog. https://blog.google/innovation-and-ai/models-and-research/google-labs/jules-now-available/ (accessed 2026-02-09).

[14] Tejun Heo. 2015. Control Group v2. Linux Kernel Documentation. https://docs.kernel.org/admin-guide/cgroup-v2.html.

[15] Tejun Heo, David Vernet, and Josh Don. 2023. Extensible Scheduler Class. Linux Kernel Documentation. https://docs.kernel.org/scheduler/sched-ext.html.

[16] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations (ICLR)*. Oral. arXiv:2310.06770.

[17] Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. 2025. The Cost of Dynamic Reasoning: Demystifying AI Agents and Test-Time Scaling from an AI Infrastructure Perspective. arXiv preprint arXiv:2506.04301.

[18] Kubernetes Community. 2024. In-Place Update of Pod Resources. Kubernetes Enhancement Proposals. https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/1287-in-place-update-pod-resources.

[19] Kubernetes Community. 2024. Pod Quality of Service Classes. Kubernetes Documentation. https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/.

[20] Kubernetes Community. 2024. Vertical Pod Autoscaler. Kubernetes Documentation. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler.

[21] Marco Laju, Donghyun Son, Saurabh Agarwal, Nitin Kedia, Myungjin Lee, Jayanth Srinivasa, and Aditya Akella. 2026. Nalar: An Agent Serving Framework. arXiv preprint arXiv:2601.05109.

[22] Hanchen Li, Qiuyang Mang, Runyuan He, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. 2025. Continuum: Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live. arXiv preprint arXiv:2511.02230.

[23] Linux Kernel Community. 2024. eBPF Verifier. Linux Kernel Documentation. https://docs.kernel.org/bpf/verifier.html.

[24] Linux Kernel Community. 2024. libbpf Overview. Linux Kernel Documentation. https://docs.kernel.org/bpf/libbpf/libbpf_overview.html.

[25] Kai Mei et al. 2024. AIOS: LLM Agent Operating System. arXiv preprint arXiv:2403.16971.

[26] Meta/Facebook. 2024. oomd: A Userspace Out-of-Memory Killer. GitHub. https://github.com/facebookincubator/oomd.

[27] Hongqiu Ni, Jiabao Zhang, Guopeng Li, Zilong Wang, Ruiqi Wu, Chi Zhang, and Haisheng Tan. 2025. Astraea: A State-Aware Scheduling Engine for LLM-Powered Agents. arXiv preprint arXiv:2512.14142.

[28] OpenAI. 2025. Introducing Codex. OpenAI. https://openai.com/index/introducing-codex/ (accessed 2026-02-09).

[29] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Ben Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload Autoscaling at Google Scale. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. ACM.

[30] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cober, Esdras Laureano, Christos Trespass, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. USENIX, 205–218.

[31] systemd Project. 2024. systemd-oomd.service — A Userspace Out-Of-Memory (OOM) Killer. systemd Documentation. https://www.freedesktop.org/software/systemd/man/systemd-oomd.service.html.

[32] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*. ACM.

[33] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. 2024. A Survey on Large Language Model Based Autonomous Agents. *Frontiers of Computer Science* 18, 6 (2024), 186345. doi:10.1007/s11704-024-40231-1

[34] 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. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *International Conference on Learning Representations (ICLR)*. arXiv:2407.16741.

[35] Johannes Weiner. 2018. PSI — Pressure Stall Information. Linux Kernel Documentation. https://docs.kernel.org/accounting/psi.html.

[36] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. ACM, 609–621. doi:10.1145/3503222.3507731

[37] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*. arXiv:2405.15793.

[38] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*.

[39] Yusheng Zheng, Yanpeng Hu, Tong Yu, and Andi Quinn. 2025. AgentSight: System-Level Observability for AI Agents Using eBPF. arXiv preprint arXiv:2508.02736.

[40] Yusheng Zheng, Yiwei Yang, Maolin Chen, and Andrew Quinn. 2024. Kgent: Kernel Extensions Large Language Model Agent. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions (eBPF '24)*. ACM, 30–36. doi:10.1145/3672197.3673434

[41] Hui Zhu. 2026. mm: memcontrol: Add BPF hooks for memory controller. LWN.net. RFC PATCH bpf-next v3 00/12, Jan 23 2026. https://lwn.net/Articles/1055698/.