AgentCgroup

AI Agent Resource Characterization and OS-Level Control

Yusheng Zheng (yzhen165@ucsc.edu)
PhD Student

Roadmap

- Motivation & Problem
- Resource Characterization (144 tasks, 2 models)
- Agent vs. Cloud Workloads & Three Mismatches
- AgentCgroup: eBPF + Bash Wrapper + Negotiation
- Evaluation & Takeaways

Motivation

AI coding agents such as **Claude Code** and **Codex** are being deployed at scale, autonomously running compilers, test suites, and package managers in sandboxed containers.

New Workloads: AI Agents

- LLM reasons → calls tools → observes → iterates
- Agent **autonomously decides** what to run and when
- Perform actions 24/7; tool usage may be heavy and bursty

…How much does it actually cost?

- People host **many concurrent agents** on shared infrastructure
- Are traditional **containers** and **VMs** enough? Do we need new tools?
- Most studies focus on safety, but what about **resource usage** and **performance**?

Characterization: Setup

144 SWE-rebench tasks, using Claude Code with two LLM backends

Backend	Execution	Tasks
GLM-4.7-Flash	Local GPU	111
Claude Haiku 4.5	Cloud API	33

Hardware: Intel Core Ultra 9 285K, 24 cores, 128 GB DDR5

Environment: Podman containers, Linux 6.15, 1s sampling

Cross-model comparison: 33 overlapping tasks

What We Measure

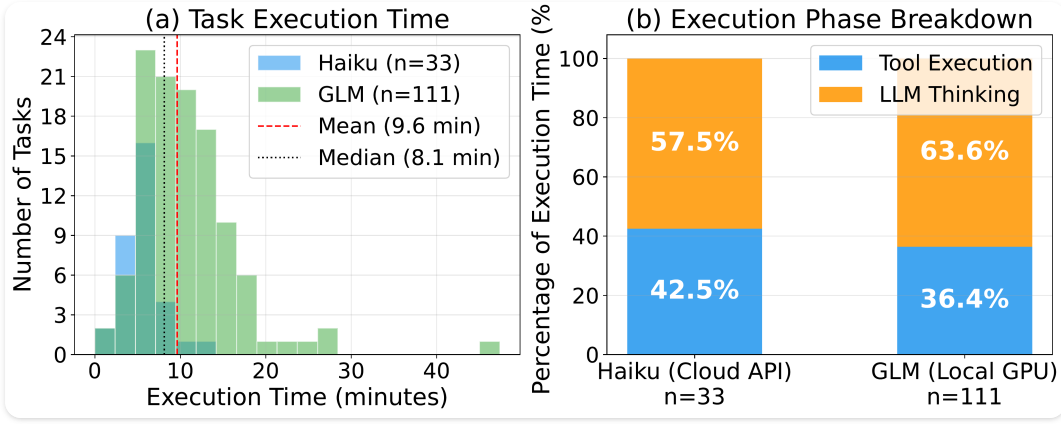
- **What does the agent do?** (execution model)
- **What does resource usage look like?** (resource dynamics)
- **Can demands be predicted?** (cross-task variance)

Container Images

- **Average: 3.5 GB, 7× larger than microservices and 70× larger than FaaS**

Execution Model

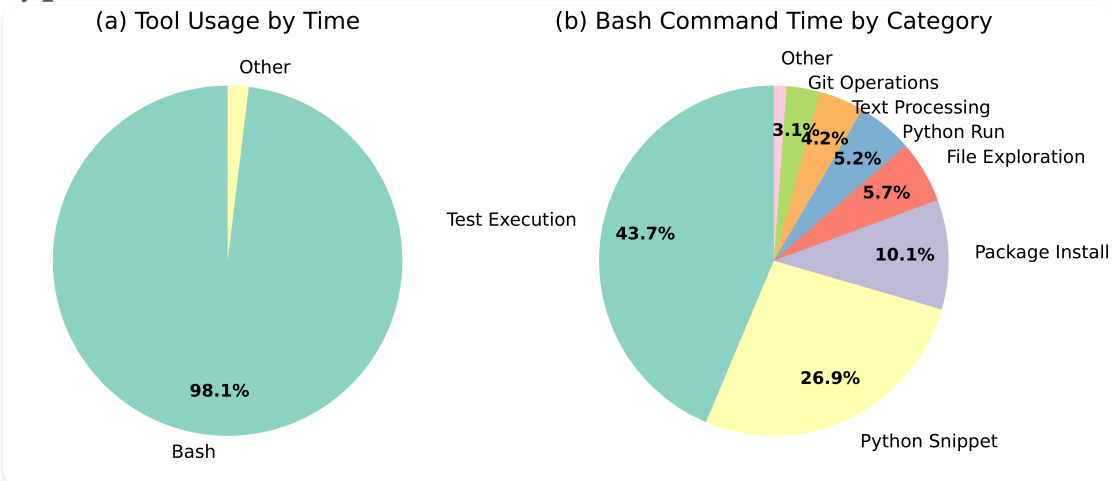
Q: Where does the time go?



- Task duration: mean 9.6 min
- LLM reasoning: 26–44% of end-to-end task latency
- The remainder:
  - tool execution (~40% of active time)
  - setup env and initialization (29–45%)

What Tool Dominates?

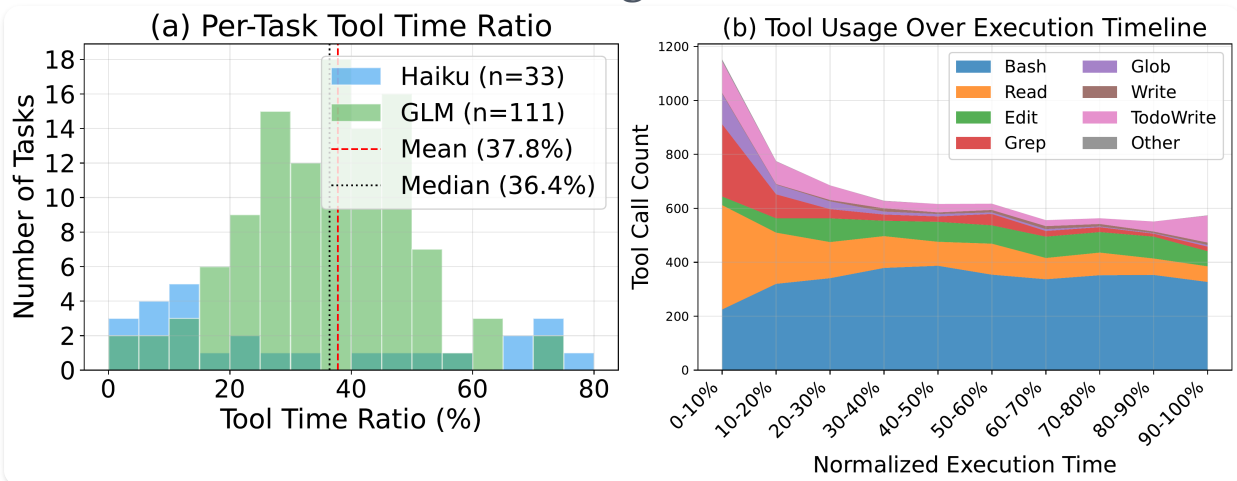
Q: Which tool types consume the most execution time?



- Bash dominates tool execution
- Top bash categories: test execution, Python snippet, package install

Tool Execution: Temporal Pattern

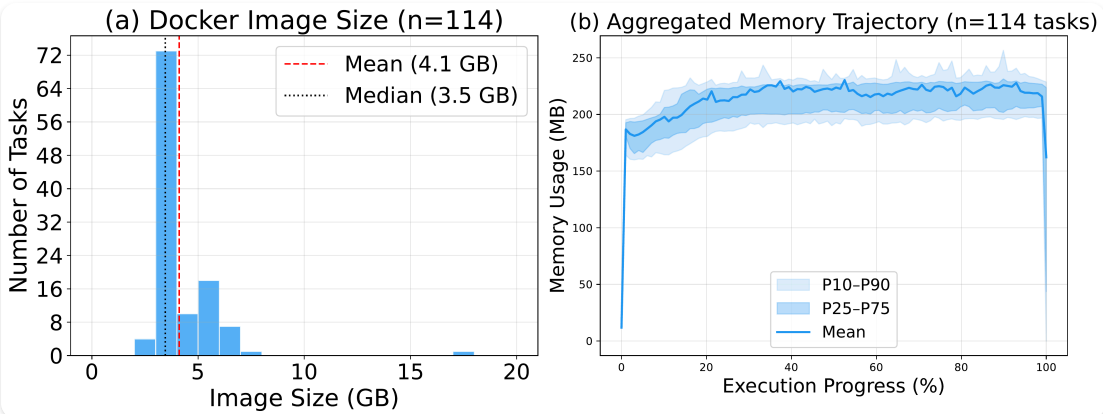
Q: When do different tools execute during a task?



- Tool calls follow an "**understand, modify, verify**" pattern: reads dominate the first 30%, and Bash peaks at 40–80%

Memory & Resource Structure

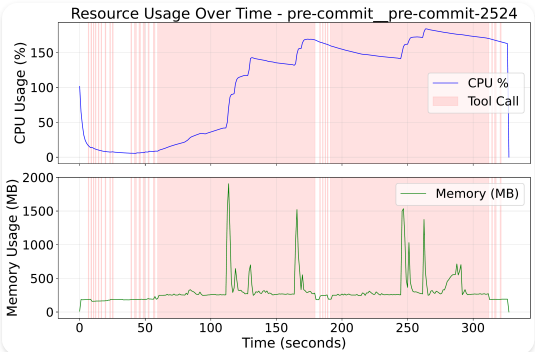
Q: What does resource consumption look like?

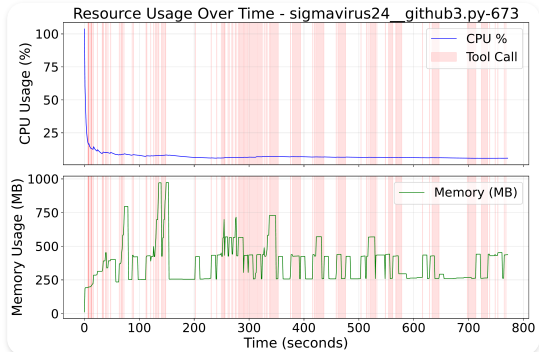


- **Memory, not CPU**, is the primary bottleneck (CPU avg <13%, memory peaks 2–4 GB)
- Resource consumption: **a ~185 MB framework baseline plus tool-call bursts**

Resource Time Series

Q: What drives resource bursts?

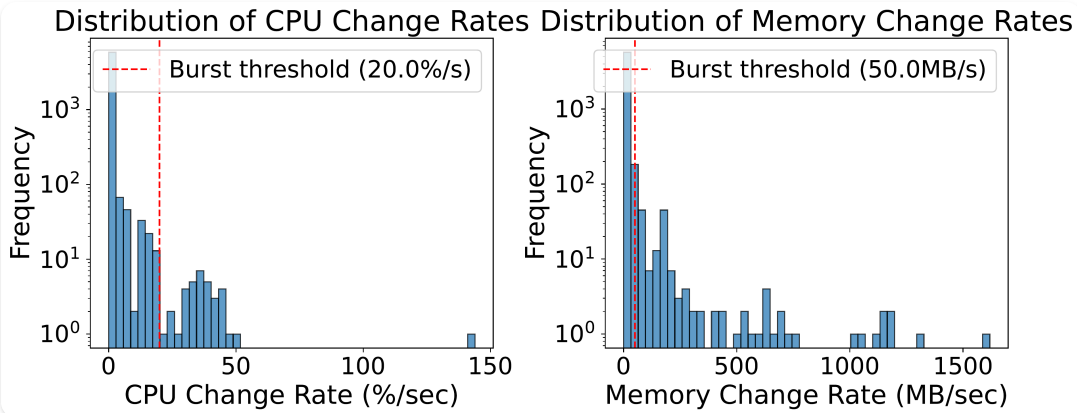




- Resource consumption is determined by what the tool *does* (e.g., pytest vs. git status), not which tool is invoked
  - **Bash calls differ by 13.7× in peak memory**
- **98.5% of memory bursts** occur during **tool calls**

Burst Dynamics & Variability

Q: How fast and how variable are resource changes?



- Resource bursts last **1–2 seconds**
- Peak-to-average ratio up to **15.4×**
- CPU-memory correlation varies by task **(−0.84 to +0.50); co-directional change cannot be assumed**

Characterization Summary

1. **OS execution dominates**: 56–74% of latency is tool execution + initialization, not LLM
2. **Memory is the bottleneck**: CPU avg <13%, but memory peaks 2–4 GB
3. **Demands are unpredictable**: 20× across tasks, 1.8× across runs; CPU/memory are decoupled

- **System optimization** is important as LLMs get faster!
- Resource management for AI agents must operate at **tool-call granularity**

Agent vs. Cloud Workloads

Q: How different from existing cloud workloads?

- **Longer-lived than serverless**: tasks last **5–11 minutes**, not **100 ms to 2 s**
- **More memory-bursty**: up to **15.4× peak/avg**, vs. roughly **1.5×–3×** in prior cloud studies
- **Less CPU-bound than batch, but memory-constrained**: CPU averages **<13%**, while memory peaks reach **2–4 GB**
- **Non-deterministic and stateful**: the same task varies **1.8× across runs**, harder to restart/migrate

Three Mismatches

Q: Why do existing resource management tools fail?

	Static Limits	Reactive Control	Predictive Scaling
Tools	mem.max/high, cpu.max; K8s QoS	PSI; oomd; TMO	VPA; Autopilot
Assumes	Known peak; stable demand	Gradual pressure; kill OK	Repeatable; history valid
Agent	15.4× peak/avg; tool-semantic	1–2s burst; unpredictable	1.8× variance; kill = lose context
Mismatch	**Granularity**	**Responsiveness**	**Adaptability**

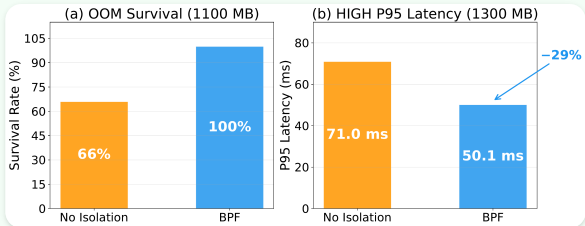
- **Granularity**: Too coarse-grained; `memory.max` at peak wastes >90%, while at average it causes OOM
- **Responsiveness**: Too slow; 3 GB/s bursts are too fast for user-space tools (tens of ms reaction time)
- **Adaptability**: History is unreliable, and generic policies cannot use agent intent

AgentCgroup Design and Preliminary Eval

Design

- Fine-grained resource domains to separate the framework from tool calls (Bash wrapper)
- Using eBPF for in-kernel enforcement (cgroup hooks)
- Intent-driven resource allocation: agents declare what they need via config

Replay Trace for Evaluation



- Fewer OOM events for multi-tenant agents
- Programmably throttle → freeze
- Negligible overhead

Limitations & Future Work

Limitations

- Requires patched kernel for prototype (`memcg_bpf_ops` under review)
- Limited evaluation
- Agents and tools are on one machine

Future Work

- An LLM inside the OS: let it understand what the agent does and negotiate with the workload
- Separate the tool environment and agent framework onto different machines (like OpenClaw)
- Evaluate on more workloads

Thank You

Questions?

References

AgentCgroup Paper (arXiv:2602.09345) · AgentCgroup GitHub