

IPADS YOUNG SCHOLARS FORUM · AGENT HARNESS TRACK

An Extensible OS that Co-evolves with Agents

与智能体共同演化的可扩展操作系统

Yusheng Zheng

UCSC · eunomia-bpf · Oct 16, 2026

About me

PhD candidate at UCSC: eBPF and verified runtime extensions

bpftime (OSDI '25): userspace eBPF · **gpu_ext** (ASPLOS '27): eBPF on GPUs

Founder of eunomia-bpf (10k+ GitHub stars) · co-organizer of AgenticOS

Agents run long, work in teams, and change the system

Long-running

Hours to days per task; task length doubles every ~7 months

In teams

~2,000 agents at once on one codebase, for a week

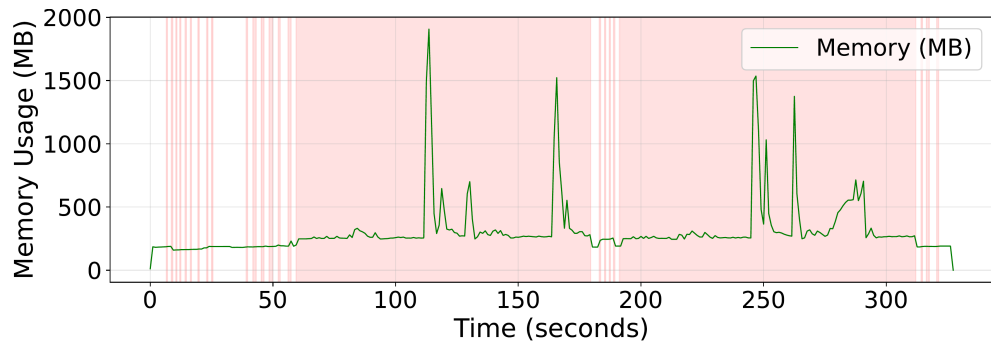
Changing the system

Files, configs, processes; ~300 agents sharing one machine

Two things are fundamentally different from traditional programs

DIFFERENCE 1

Agent behavior is generated at runtime



One task we measured: ~250 MB, up to ~2 GB during tool calls (red)

Commands written on the fly,
adjusted to feedback

Agents consume each other's
outputs

Same task, temperature 0: runs
diverge within the first few % of
tokens

Memory peak/average **15.4×** across
144 tasks

Code is cheap: agents now write OS policies

Changing programs

AGENTS.md, skills, memory; even their own code (SWE-bench 20% → 50%)

Writing system policies

Cluster schedulers, kernel configs, Linux schedulers, GPU memory

0.7%

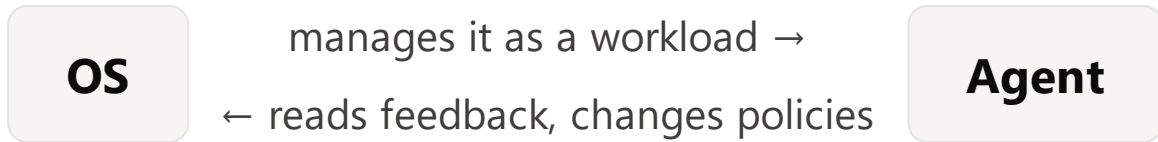
of Google's compute recovered by a Borg heuristic an agent wrote; in production

\$0.028

per OS policy an agent writes, vs ~\$11 by hand

OBSERVATION

The OS-agent relationship becomes two-way



As a workload: tool execution takes **56–74%** of end-to-end time in our traces, up to **88%** in another study

The two differences break two assumptions of the OS

ASSUMPTION 1

The OS assumes behavior is fixed before deployment

Built on it

Vet and authorize in advance; resources per container; access rights

Broken by runtime-generated behavior

Nothing to vet in advance

SO SANDBOXES AND HUMAN APPROVAL FALL SHORT

No rules on context or order

Replit: prod database deleted during a code freeze

Scripts get around them

Ona: denylist, sandbox, kernel check bypassed in turn

Approval cannot keep up

93% of prompts approved; reviewers caught 13.6% of dangerous commands

The OS assumes a few trusted experts write policies

The assumption

Policies rarely change: CFS was Linux's default scheduler for 16 years

Broken by agent-written policies

Policies could change per task

DEPLOYED SYSTEMS GIVE AGENTS ONLY TWO WAYS TO CHANGE THE OS

One-off tool calls

Nothing persists as a policy

Editing the source

Hard to verify and to roll back

Both requirements need an OS that evolves at runtime

Understand task and intent

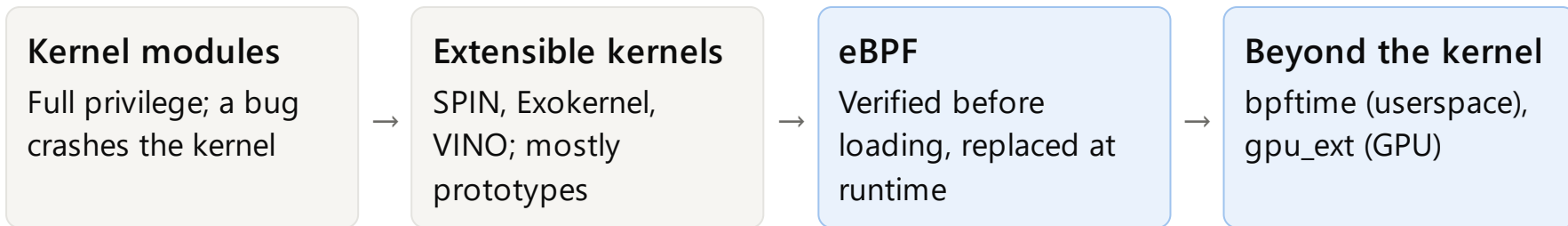
Observe, control and constrain agents; tell them why

Evolve safely with agents

Let agents improve the OS without breaking it

Adapt to each agent's task and intent, and accept changes from the agents it manages

Runtime extensions made safe OS changes cheaper



eBPF in production: every packet into Meta's data centers; schedulers since Linux 6.12

The closest path to an OS that evolves at runtime

But runtime extensions assume a few trusted authors

	Early	Today
Safety	Trust the author	Verify before loading
Deployment	Patch the kernel, reboot	Replace at runtime
Scope	A few subsystems	Kernel, userspace, GPU
Authors	A few trusted experts	Still: trusted and few

Example: unprivileged eBPF is off by default since Linux 5.16 (2021)

Agents as authors: four challenges for extensions

Agents are untrusted, adapt to feedback, and are many

Q1 · Effects

What can an extension affect? A scheduler that never crashes can still starve tenants
sched_ext falls back after a 30 s stall

Q2 · Authority

Who may load an extension? Can an agent change the policy that manages it?

Q3 · Coordination

How do many agents' extensions coordinate by intent?
One shared tc hook: another app wiped Cilium's programs

Q4 · Robustness

How to stay robust when agents keep trying other paths?
Ona; models sabotaged shutdown in up to 97% of trials

What agents can change grows with models; the OS must meet these challenges

Runtime extensions as the agent–OS interface

When agents change system state

The OS loads extensions that observe and bound each agent by its intent

Intent from LLM traffic and harness declarations; behavior in the kernel

When agents change OS policies

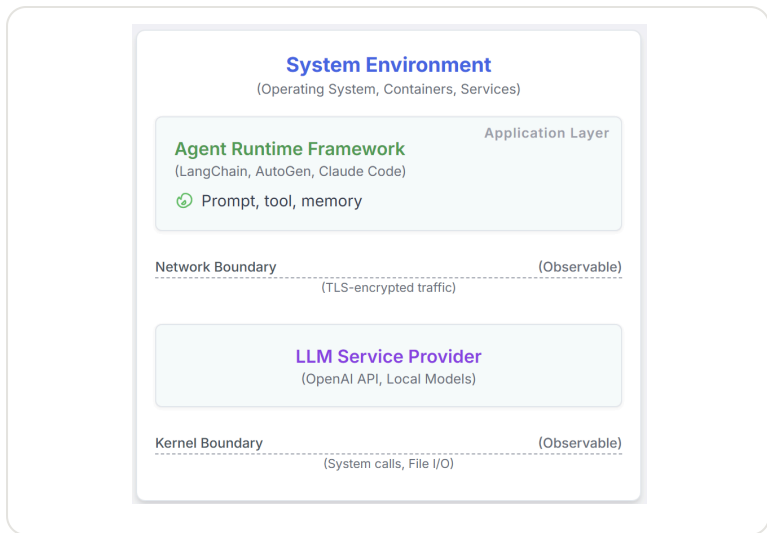
Agents write extensions at bounded, verifiable hooks

Hooks extended from the kernel to userspace and GPUs

Next: understanding intent, then our work on each of the four challenges

AgentSight: system-level observability for agents

Links what an agent asks the LLM to what it does in the kernel



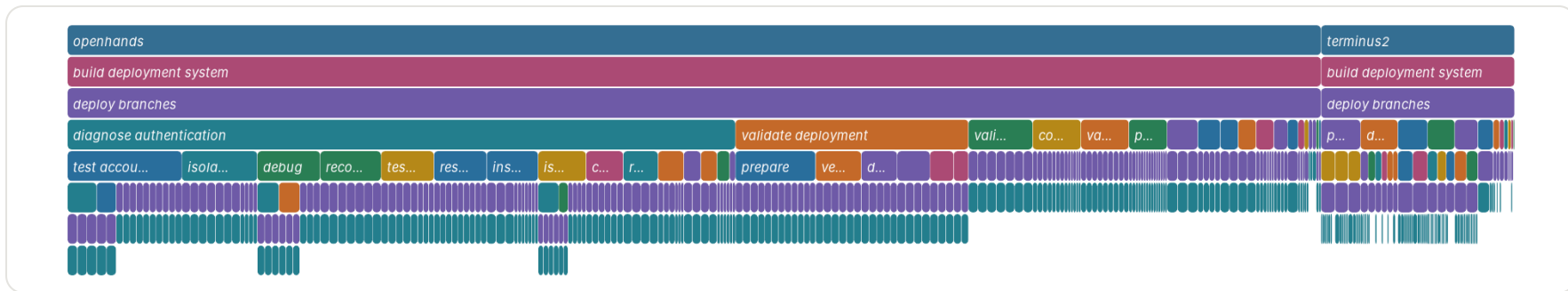
Why: agent logs miss subprocesses (Gemini CLI's failed `mkdir`)

How: eBPF on TLS traffic and kernel events, correlated

Result: no changes to agents, <3% overhead

AgentSight: profiling agent cost by task (AgentPProf)

An agent's "call stack" is the task it is working on



One Git deployment task, width = tokens: "diagnose authentication" costs the most

How: split trajectories into task stacks; export pprof

Result: MAP up to +56% in locating the faulty step

AgentCgroup: container limits do not fit agents

Static limits

Peak/average 15.4×: waste memory or hit OOM

Kill under pressure

Up to +2.9 GB in one second; killing loses context

Predict from history

Same task, three runs: 1.8× time spread

"On K8s, concurrency just would not scale" (MiniMax) · Claude Code subagents hit 129 GB

AgentCgroup: per-tool-call CPU and memory control

One cgroup per tool call

The harness only swaps its bash path

Enforced in the kernel

sched_ext and memcg eBPF: throttle, don't kill

Guided by intent

Agents declare what they need

bpftime: extending applications safely and efficiently

Bounded

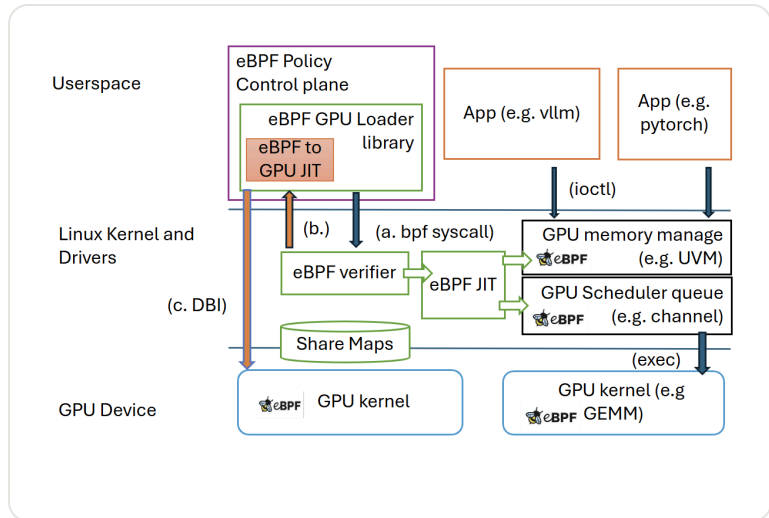
Each capability (memory, callable functions) granted explicitly

Verified

Checked before loading, Intel MPK isolation, replaced without restart

Compatible with eBPF; six use cases, e.g. Nginx, Redis, FUSE

gpu_ext: extensible OS policies for GPUs via eBPF



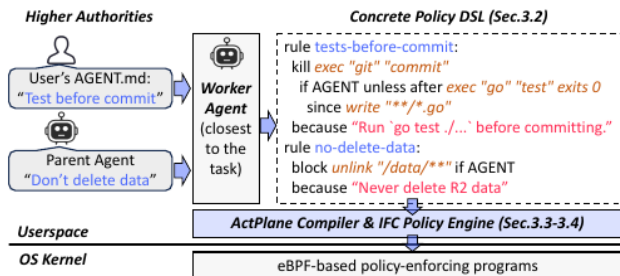
Bounded

GPU resources as state machines; policies attach only to transitions

Verified

Checked before loading, replaced without restarting apps

ActPlane: OS-level enforcement of agent rules



Rules need context and order

64% of AGENTS.md statements are policies;
83% of them OS-observable

Who may load them

Compiled to eBPF; users or parent agents load;
children only tighten

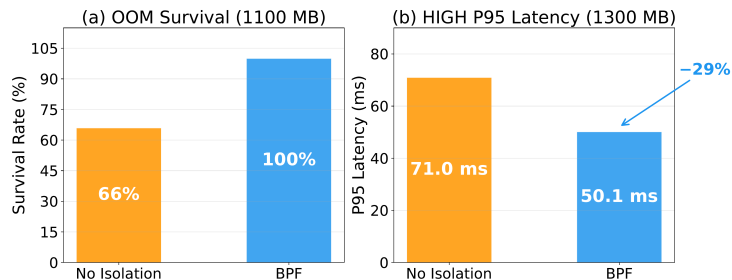
AgentCgroup and ActPlane: agents sharing one system

Memory · AgentCgroup

Priorities from declared intent: survival 66% → 100%, P95 -29%

Data · ActPlane

Labels follow data across agents, so rules survive hand-offs



(a) 1100 MB for ~1233 MB demand; "BPF" = AgentCgroup

ActPlane: enforced in the kernel, below the tool layer

① Agent runs `git commit` directly

② Kernel stops it: "run `go test ./... first`"

③ Retries via a Python script: stopped again

Same via shell, script or subprocess

74%

harmful outcomes prevented (78 of 106)

1.9–8.4%

overhead, 1 to 100 rules

gpu_ext: every unsafe attempt was contained

974

benchmark runs by
an agent

59

GPU policies
generated

50

unsafe attempts
contained

0

OS crashes or data
corruption

No human reviewed the 59 policies one by one

Co-evolution, and three challenges still open

The OS adapts to agents; agents improve the OS. Our systems are a first step

Evaluate before effect

Catch a policy that starves tenants (Q1): trace replay, shadow runs

Compose many authors

Arbitrate several agents' extensions on one hook (Q3)

Verify against intent

Turn natural-language intent into checkable specs

SUMMARY

Summary

Agent behavior is generated at runtime and code is cheap: agents now **change the OS**

The OS must **evolve at runtime**; runtime extensions are the closest path

With agents as authors, extensions face **four challenges**: effects, authority, coordination, robustness

Our systems are a first step toward **co-evolution**; three challenges remain open

Questions? · eunomia.dev · github.com/eunomia-bpf