Towards Agentic OS: An LLM Agent Framework for Linux Schedulers

Abstract

Operating system schedulers suffer from a fundamental semantic gap, where kernel policies fail to understand application-specific needs, leading to suboptimal performance. We introduce SchedCP, the first framework that enables fully autonomous Large Language Model (LLM) agents to safely and efficiently optimize Linux schedulers without human involvement. Our core insight is that the challenge is not merely to *apply* a better LLM, but to architect a decoupled control plane that separates the AI's role of semantic reasoning ("what to optimize") from the system's role of execution ("how to observe and act"). Implemented as Model Context Protocol(MCP) server, SchedCP provides a stable interface with three key services: a Workload Analysis Engine, an evolving Scheduler Policy Repository, and an Execution Verifier that validates all AI-generated code and configure before deployment with static and dynamic analysis. We demonstrate this architecture's power with sched-agent, a multi-agent system that autonomously analyzes workloads, synthesizes custom eBPF scheduling policies, and deploys them via the sched_ext infrastructure. Our evaluation shows that SchedCP achieves up to 1.79x performance improvement and 13x cost reduction compared to naive agentic approaches, all while maintaining high success rate. The code will be open-sourced.

1 Introduction

Operating system schedulers face a fundamental challenge: kernel policies cannot understand what applications need, leading to suboptimal performance as Linux's EEVDF scheduler [9] applies one-size-fits-all policies to diverse workloads. While sched_ext [6] in Linux 6.12 enables custom extended Berkeley Packet Filter(eBPF) schedulers with safety guarantees through verification, developing them still requires both deep kernel expertise and a good understanding of the workloads.

Prior reinforcement learning-based schedulers [10, 17] lack semantic understanding of workloads. While LLMs [15, 1] and agent frameworks [18, 14, 4, 16, 8] excel at code generation, naively applying them to scheduler development proves impractical: our experiments show generating a basic scheduler takes 33 minutes, costs \$6, and often degrades performance. The gap remains: existing methods lack semantic understanding, while LLMs lack the scaffolding for safe, efficient, and reliable systems integration despite prior LLM-eBPF work [20].

We introduce a decoupled architecture with two components. First, SchedCP is a control plane framework providing safe AI-kernel interfaces with profiling, tracing and validation tools, allowing LLMs to customize the kernel scheduler with eBPF. It exposes kernel scheduling features via Model Context Protocol (MCP) through three core services: Workload Analysis Engine, Scheduler Policy Repository, and Execution Verifier. Second, sched-agent is the first autonomous multi-agent system that decomposes scheduler optimization into four specialized agents (Observation, Planning, Execution, Learning), demonstrating how LLMs can bridge the semantic gap between application requirements and kernel scheduling policies. This separation allows SchedCP to provide a generalizable framework for any AI agent, while sched-agent demonstrates semantic workload analysis and policy generation. Our evaluation shows sched-agent achieves up to 1.79× performance

on kernel compilation, 2.11× P99 latency improvement and 1.60× throughput gain on schbench, 20% latency reduction for batch workloads, and 13× cost reduction.

2 Motivation

Linux scheduler optimization faces three barriers. First, a domain knowledge gap exists between developers and users: DevOps engineers lack insight into workload characteristics (latency-sensitive vs. throughput-oriented), while edge/personal device users lack both kernel optimization expertise and understanding of application-specific targets. Second, scheduler development requires mastering kernel programming, limiting innovation to few experts. Third, modern workloads exhibit complex dynamics: web traffic varies by orders of magnitude daily, build system parallelism changes with dependencies. Prior RL-based schedulers [10, 17, 19, 11] require extensive training per workload type, lack semantic understanding to transfer across workloads, and cannot generate code. LLMs uniquely bridge these gaps by: (1) understanding natural language and source code semantics without task-specific training, (2) synthesizing correct eBPF schedulers based on automatic workload characterization, (3) reasoning about performance trade-offs and system constraints, (4) LLM can operate in the control plane to generate optimized code that runs natively with negligible runtime overhead, unlike traditional ML models that would cause unacceptable inference latency in the scheduler hot path.

We tested Claude Code[4], the state of the art LLM agent, with "write a FIFO scheduler in eBPF" from an empty folder, with all permissions and bash access. Of three attempts, only one succeeded. The second attempt produced pseudo-code after 6 minutes trying, and the third generated a scheduler tracer instead after 8 minutes development. The successful generation required 33 minutes, 221 LLM API calls, and 15+ iterations, costing $6 (vs. 5 minutes typically for an expert developer). The generated code, for some workloads, exhibited poor quality with excessive overhead, performing worse than EEVDF. The agent required root access, could crash the system during testing, and lacked fallback mechanisms, which also raises safety concerns. These experiments reveal three critical challenges: **Performance**, ensuring AI schedulers outperform existing ones; **Safety**, preventing crashes, lockups, and starvation while minimizing privileges; and **Efficiency**, reducing the 33-minute, $6 generation cost for practical deployment.

3 The SchedCP Framework Design and Implementation

As illustrated in Figure 1, SchedCP is a secure control plane acting as an 'API for OS optimization,' separating systems infrastructure from AI logic, distinguishing "what to optimize" (AI's domain) from "how to observe and act" (system's domain). Four key principles govern its design. First, **decoupling and role separation** ensures future-proofing by treating the AI agent as a performance engineer using a stable set of tools. Second, **safety-first interface design** addresses the inherent risks of autonomous agents with kernel access by treating AI as potentially non-cautious actors, designing defensive interfaces that prevent catastrophic failures by default, and avoiding granting 'root' privileges. Third, **adaptive context provisioning** addresses LLM agents' constraints from finite context windows and token costs: agents start with minimal summaries and progressively request details as needed. Finally, **composable tool architecture** follows Unix philosophy by providing atomic tools that let agents construct complex workflows through their reasoning capabilities, enabling novel solution generation rather than constraining them with rigid workflows. Implemented in 4000 lines of Rust and 6000 lines of Python (including tests), SchedCP provides essential tools for any agent to interact with the Linux kernel's scheduler, analogous to how an environment in reinforcement learning provides state, actions, and rewards for learning. It exposes its services via the Model Context Protocol (MCP) [2] through three primary services:

1. Workload Analysis Engine Provides tiered access to system performance data: (1) cost-effective API endpoints with pre-processed summaries (CPU load, memory usage), (2) secure sandbox access to file reading, application building, Linux profiling tools (`perf`, `top`) and dynamically attachable eBPF probes, (3) feedback channel reporting post-deployment metrics (percentage change in throughput/latency).

2. Scheduler Policy Repository. Database storing executable eBPF scheduler programs with metadata (natural language descriptions, target workloads, historical performance metrics). It provides
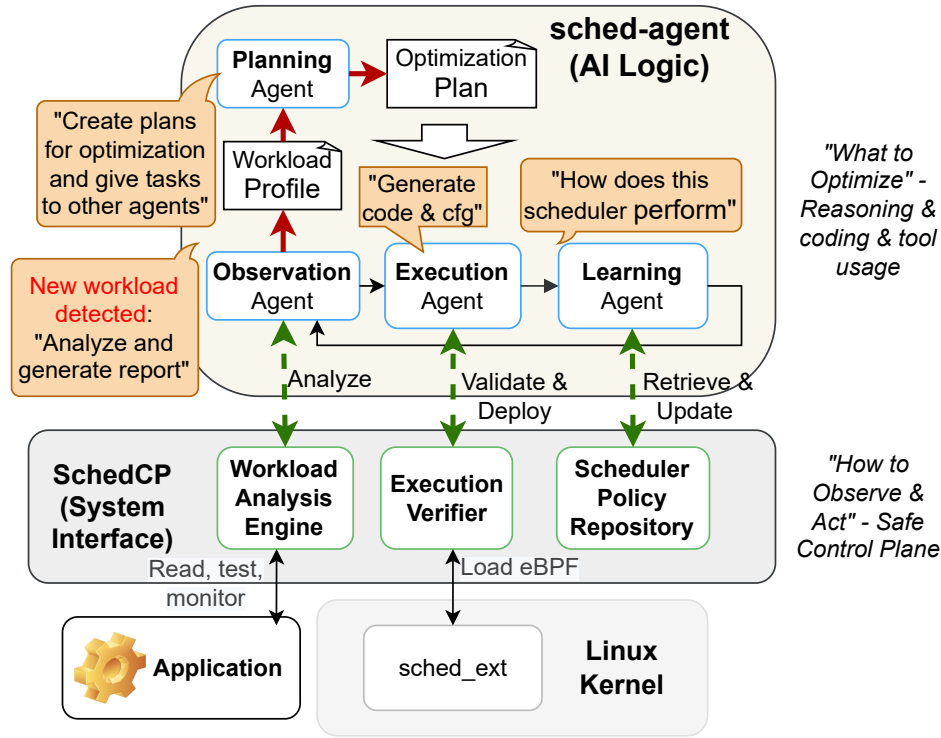
Figure 1: **Overall architecture of SchedCP and sched-agent.** SchedCP (bottom) provides the system interface with three core services: Workload Analysis Engine, Scheduler Policy Repository, and Execution Verifier. sched-agent (top) implements the AI logic through four specialized agents (Observation, Planning, Execution, Learning) working in a closed loop. Red lines show initialization when detecting new workloads, black arrows show optimization loops, and green arrows indicate tool usage by agents.

APIs for semantic search and retrieval, enabling agents to find relevant schedulers or composable code primitives. To support system evolution, it includes endpoints for updating performance metrics and promoting new policies, reducing generation costs by allowing reuse of proven solutions while maintaining a growing library of scheduling strategies.

3. Execution Verifier includes a multi-stage validation pipeline: (1) kernel's eBPF verifier ensures memory safety and termination, (2) scheduler-specific static analysis detects logic flaws (starvation, unfairness) the standard verifier misses, (3) dynamic validation in secure micro-VM tests correctness and performance. Successful validation issues signed deployment tokens for monitored canary deployments with circuit breakers to revert if performance degrades, eliminating sched-agent's need for root access.

4 sched-agent: A Multi-Agent Framework for OS Optimization

Building on SchedCP, we developed **sched-agent**, a multi-agent AI framework implementing in-context reinforcement learning (ICRL)[13] for scheduler optimization. Using Claude Code's subagent architecture[5], sched-agent decomposes optimization into four distinct ICRL stages through specialized AI assistants with customized prompts, tools, and separate context windows[3]. The framework integrates with container orchestrators (Kubernetes, Docker) to automatically trigger optimization when applications deploy, enabling adaptive strategy refinement based on performance feedback without model retraining.

The **Observation Agent** builds Workload Profiles by querying the Workload Analysis Engine strategically, starting with high-level summaries from process name, commands then requesting deeper profiling (`perf stat`, `top`) based on findings, synthesizing data into natural language descriptions and optimization goals while managing cost-precision tradeoffs. For kernel compilation, it produces profiles like "CPU-intensive parallel compilation with short-lived processes, inter-process dependencies, targeting makespan minimization." The **Planning Agent** transforms profiles into optimization strategies via the Scheduler Policy Repository, following a decision hierarchy: configuring existing schedulers, generating patches, or composing new schedulers from primitives. The **Execution Agent** manages development, validation and deployment by synthesizing code artifacts, submitting to the Execution Verifier, interpreting results to refine code or fix logic issues. The **Learning Agent** completes the ICRL loop by analyzing deployment outcomes (e.g., 45% makespan reduction), enabling in-session adaptation and updating the repository with refined metrics, deployment contexts, and documented antipatterns.

5 Evaluation

We validate SchedCP's effectiveness through four research questions: configuring existing schedulers (RQ1), generating new schedulers for specific workloads (RQ2), cost and efficiency of scheduler generation (RQ3), and iterative refinement improvements (RQ4). Evaluation uses two machines: 86-core Intel Xeon 6787P with 758GB RAM running Linux 6.14, and 8-core Intel Core Ultra 7 258V with 30GB RAM running Linux 6.13. Agents use Claude Code (Opus 4), testing each case three times and averaging results. All experiments successfully create working custom scheduler configurations or eBPF programs.

Scheduler Configuration: For kernel compilation (tinyconfig, "make -j 172" on 6.14 source), SchedCP achieves 1.63× speedup with scx_rusty initially, then iterative refinement selects scx_layered for 16% additional gain, reaching 1.79× total improvement over EEVDF (Figure 2a). Pre-trained RL approaches [7] show no improvement, likely because they require costly hardware/workload-specific retraining. On schbench [12], initial AI configuration (scx_bpfland) underperformed, but three refinement iterations identified scx_rusty as superior: 2.11× better P99 latency and 1.60× higher throughput versus EEVDF (Figure 2b), demonstrating effective learning from feedback.

(a) AI configured scheduler for Linux build time

(b) AI configured scheduler for Schbench latency and throughput

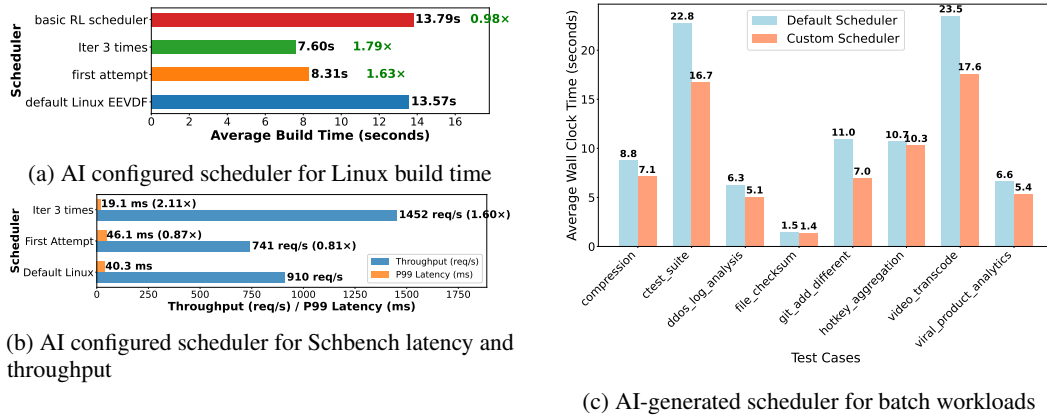(c) AI-generated scheduler for batch workloads

Figure 2: Performance evaluation of SchedCP across different workloads

New Scheduler Synthesis: For 8 diverse batch workloads (file compression, video transcoding, software testing, data analytics) with long-tail distributions (40 parallel tasks: 39 short, one long), sched-agent correctly identified the optimization goal and workload pattern, implementing Longest Job First (LJF) scheduling to achieve 20% average latency reduction (Figure 2c). Claude Opus successfully classified all 8 workloads at $0.15 per analysis, while Claude Sonnet failed. Generation efficiency improved 13× (to 2.5 minutes) with $0.45 synthesis cost per workload, demonstrating economic viability alongside performance gains.

References

[1] Anthropic. The claude 3 model family: Opus, sonnet, haiku. *Anthropic Technical Report*, 2024.

[2] Anthropic. Model context protocol. `https://www.anthropic.com/news/model-context-protocol`, 2024. An open standard for connecting AI assistants to data sources.

[3] Anthropic. How we built our multi-agent research system. `https://www.anthropic.com/engineering/built-multi-agent-research-system`, 2025. Engineering blog post on multi-agent systems.

[4] Anthropic. Introducing claude code. `https://www.anthropic.com/news/claude-code`, Feb 2025. Agentic coding tool announcement, Anthropic blog.

[5] Anthropic. Subagents - claude code documentation. `https://docs.anthropic.com/en/docs/claude-code/sub-agents`, 2025. Documentation for Claude Code subagent implementation.

[6] Linux Kernel Community. sched_ext: Bpf scheduler class. *Linux Kernel Documentation*, 2024.

[7] Jonathan Corbet. Improved load balancing with machine learning. *LWN.net*, July 2025. Machine learning approaches to Linux kernel scheduling.

[8] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, et al. MetaGPT: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[9] Linux Kernel Documentation. Eevdf scheduler - earliest eligible virtual deadline first. `https://docs.kernel.org/scheduler/sched-eevdf.html`, 2024. Introduced in Linux kernel 6.6, replacing CFS.

[10] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 270–288, 2019.

[11] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Park: An open platform for learning-augmented computer systems. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[12] Chris Mason. schbench: Scheduler benchmark. `https://kernel.googlesource.com/pub/scm/linux/kernel/git/mason/schbench`, 2016. A scheduler benchmark that measures wakeup latency and throughput.

[13] Amir Moeini, Jiuqi Wang, Jacob Beck, Ethan Blaser, Shimon Whiteson, Rohan Chandra, and Shangtong Zhang. A survey of in-context reinforcement learning. *arXiv preprint arXiv:2502.07978*, 2025. Version 1, February 11.

[14] Taylor Mullen and Ryan J. Salva. Gemini cli: Your open source ai agent. `https://blog.google/technology/developers/introducing-gemini-cli-open-source-ai-agent/`, 2025. Google Developers Blog, Jun 2025.

[15] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[16] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, et al. ChatDev: Communicative agents for software development. In *Proc. ACL*, 2024.

[17] Haoran Qiu, Siddhartha Banerjee, Saurabh Jha, Shivaram Kalyanaraman, and Chuan Tang. Firm: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 805–825, 2020.

[18] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enable next-gen large language model applications, 2023.

[19] Wei Zhang et al. Multi-resource scheduling with reinforcement learning. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[20] Yusheng Zheng, Yiwei Yang, Maolin Chen, and Andrew Quinn. Kgent: Kernel extensions large language model agent. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 30–36, New York, NY, USA, 2024. Association for Computing Machinery.