An Expert in Residence: LLM Agents for Always-On Operating System Tuning

Georgios Liargkovas[1] **Vahab Jabrayilov**[1] **Hubertus Franke**[2] **Kostis Kaffes**[1]
[1]Columbia University [2]IBM Research
`{gliargko, vjabrayilov, kkaffes}@cs.columbia.edu`[1] `frankeh@us.ibm.com`[2]

Abstract

Classical machine-learning auto-tuners for OS control struggle with semantic gaps, brittle rewards, and unsafe exploration. We introduce an online, LLM-driven agent that emulates expert reasoning for continuous OS optimization. When tuning the Linux Completely Fair Scheduler's hyperparameters, the agent outperforms Bayesian optimization by 5% in single-parameter tuning, 7.1% in two-parameter co-tuning, and a human expert by 2.98% overall, while converging faster and adapting more quickly to workload changes. When application counters are unavailable, system-level proxies (e.g., Instructions Per Cycle (IPC)) preserved tail latency in our setup. Putting this together, we propose adopting the Model Context Protocol (MCP) for tool/resource discovery and invocation and a logging channel; on top of that, we propose adding transactional apply–commit–revert, host-mediated approval gates, and policy controls in the OS-tuning server and host to ensure safe, auditable operation. Our results and reference design suggest a practical path toward safe, self-adapting OS control.

1 Introduction

Modern operating systems expose hundreds of tunable run-time knobs, but manually optimizing them for evolving workloads and hardware is intractable. We argue that the next leap in autonomous systems management lies in emulating the expertise of a human engineer. An expert does not just blindly turn knobs; they leverage deep contextual understanding and sophisticated reasoning to make informed decisions. This paper introduces a new paradigm that embodies this principle: a fully online, autonomous agent powered by a Large Language Model (LLM) for live operating system tuning. This agent bridges the semantic gap that plagues traditional machine learning (ML) auto-tuners, replacing slow, risky exploration and brittle reward functions with an agent that can reason about system state, interpret high-level goals, and act decisively, much like a human expert.

We deploy the agent for live CPU scheduler tuning on Linux, showing it outperforms classical ML tuners and a human expert while adapting to workload shifts.

Related Work: Classical ML auto-tuners, whether based on Bayesian Optimization (BO) or Reinforcement Learning (RL), have shown success in specific domains like database systems [6, 18], application tuning [6, 10, 12, 17], system resource allocation [2], and others. However, their lack of contextual understanding makes them ill-suited for the dynamic and sensitive environment of a live OS. More recent work has begun to integrate LLMs, but primarily in offline settings. Systems like DB-BERT [16], GPTuner [8], and λ-Tune [7] use LLMs to pre-process documentation or generate static configurations before deployment. Similarly, AutoOS [3] uses an LLM to generate an optimized kernel configuration, but this is a one-off, pre-deployment step. While visionary agentic frameworks like AIOS [14] and Herding LLaMaS [9] have been proposed, they have not yet demonstrated a practical system for specialized, continuous performance tuning on a live machine.

Position & Contributions: (1) We argue for an agentic OS tuner that uses tools to build its own context and act online. (2) We propose a principled interaction model that uses MCP for discoverability and communication, and layers server/host-side transactions, approval gates, and policy to make actions governable and auditable. (3) We demonstrate a mini case study on Linux's Completely Fair Scheduler (CFS) showing faster, more stable convergence than BO/RL/human, including when co-tuning antagonistically correlated knobs (parameters can override each other).

2 Mini Case Study: LLM Tuning Loop vs Classical Tuners

We study online tuning of the Linux kernel's CFS [15]. The goal is to minimize application-level p99 tail latency under load by adjusting two related knobs: `latency_ns` (the target period for fairness) and `min_granularity_ns` (the minimum timeslice a task may run). We evaluate two scenarios: (i) a fixed-rate TPC-C workload with both single-parameter (1P) and dual-parameter (2P) tuning, and (ii) a variable-rate TPC-C workload with two mid-run rate changes (1P) which includes a human expert.

We compare four online tuning strategies: (1) Bayesian Optimization (BO), (2) Reinforcement Learning (Q-learning and DQN), (3) a human expert with kernel experience, and (4) a prompt-driven LLM loop (Gemini 2.5 Flash). All tuners run for the same number of cycles (200 by default). Each cycle consists of a 10-second workload run, metric collection, and a new configuration proposal. For (ii), the LLM and the human receive the same interface and auxiliary system metrics (e.g., IPC from `perf stat` and throughput from the application), while the classical baselines only consume the scalar reward exposed to them (p99 latency; for "IPC" the reward is IPC; details in Appendix A).

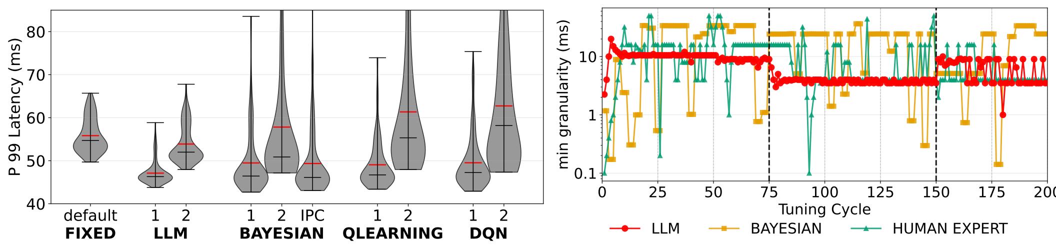

Figure 1: **Left:** Violin plots of application p99 latency (lower is better) under single- and dual-parameter tuning of the antagonistic pair (`latency_ns`, `min_granularity_ns`) in the fixed-rate experiment; red line denotes the mean. Labels "1/2" denote one- vs two-parameter tuning; FIXED is the default OS configuration. Co-tuning degrades classical tuners (wider, higher distributions), while the LLM loop maintains low, stable latency across both settings. **Right:** Variable-rate experiment (1P): evolution of `min_granularity_ns` over 10 s cycles with two mid-run rate changes (dashes). Both LLM and human avoid sustained operation below 2 ms in our setup; brief early human dips reflect coarse exploration under noise before re-stabilization, while BO settles more slowly.

Figure 1 summarizes the results. On the left, we co-tune two antagonistic knobs (2P): if `min_granularity_ns` is set too high it overrides `latency_ns`, skewing fairness and delaying short tasks; because changing one can negate the other, co-tuning is typically worse than tuning either in isolation. Consistent with this, BO and RL perform worse than the default under 2P (e.g., BO 58.09 ms vs default 55.85 ms), while the LLM still outperforms it (53.95 ms vs. 55.85 ms). The right plot tracks the evolution of `min_granularity_ns` over tuning cycles. RL methods exhibit erratic jumps and fail to converge (skipped for clarity). BO is smoother but slow to settle. In contrast, the LLM converges within 10 cycles, adjusts when workload intensity shifts, and re-stabilizes, closely mirroring the human expert but with fewer missteps.

These differences stem from core limitations in classical methods. First, without explicitly encoded priors/constraints, vanilla BO/RL in our setting may explore myopically and miss structural relationships (e.g., the latency–min_granularity dependency). Second, their engineering cost is high: reward function design, parameter discretization, and hyperparameter tuning all require deep system and ML expertise. Finally, exploration is often unsafe: BO and RL may explore pathological configurations (e.g., tiny timeslices) that hurt performance or destabilize the workload.

The LLM-based tuner overcomes these issues by emulating expert reasoning and, in practice, behaves much like our human baseline. Both the human and the LLM consider the same auxiliary metrics (throughput, IPC) and follow a similar pattern at change points: a coarse corrective move followed by small, monotonic refinements. The LLM's advantages are reaction time, smaller overshoot, and safer exploration: it avoids obviously harmful regions (e.g., `min_granularity_ns` below $\sim$2 ms in our setup) and takes fewer, smaller steps to re-stabilize.

Quantitatively, in the variable-rate experiment the LLM outperforms the human overall and in each phase: 46.24 ms vs 47.66 ms overall (+2.98%), 45.71 ms vs 47.40 ms at 300 tx/s (+3.57%), and 47.78 ms vs 48.40 ms at 1100 tx/s (+1.29%). Against Bayesian optimization in the fixed-rate setting, the LLM (Gemini 2.5 Flash) reduces p99 by 5.0% in 1-parameter tuning (47.16 ms vs 49.62 ms) and by 7.1% in 2-parameter tuning (53.95 ms vs 58.09 ms), with consistently lower variance.

Based on the above insights (details in Appendix B), we conclude that an LLM agent can offer a significant outcome advantage over classical approaches and even human experts, while simultaneously reducing the engineering effort required to build and deploy a production-ready tuner. While tightly coupled to the application in this example, below we generalize this paradigm to an autonomous agent architecture that can reason, explore, and act independently using tools and system state.

3 From Knob Tuner to Autonomous Agent

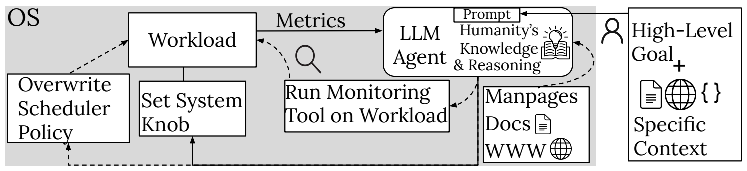

Figure 2: An autonomous LLM-powered OS tuning agent. It uses tools and resources to independently monitor the system, access knowledge, set system knobs, or generate custom OS policies.

The tuner in our case study is reactive: it waits for metrics, then proposes a change, without independent initiative. To progress toward autonomy, we are currently realizing the agentic architecture in Fig. 2: the LLM gains tool access to *observe* (e.g., `iostat`, `perf`, knob reads), *act* (e.g., knob writes, policy deployment), and *retrieve knowledge* at runtime. With self-gathered evidence, the agent moves beyond reactive tuning by forming testable hypotheses and running targeted experiments; for instance, it can detect a NUMA-sensitive workload, pin processes to a node, and adjust memory reclaim settings, then *trial–measure–commit/rollback* the change using host/server-side guardrails. We use MCP for discovery and invocation of tools and resources; tool inputs are described via JSON Schema and validated server-side (types/ranges), and we add semantic/unit checks beyond schema. We use MCP's logging channel and optional client-side elicitation for approvals. This mirrors an expert workflow and reduces prompt brittleness—minor input changes leading to significantly different LLM outputs—by grounding actions in schema-checked calls rather than free-form instructions.

While we used hosted APIs for convenience in the case study, the architecture is model-agnostic and can support on-host small/quantized models to avoid third-party SLAs on critical loops.

A key advantage is that the loop is not tied to application-specific metrics. When app counters are missing, the agent can (i) fall back to system-level proxies (e.g., IPC) and (ii) discover available app metrics by listing MCP tools/resources. In our setup, optimizing IPC produced p99 comparable to optimizing p99 directly, but proxies can be brittle across workloads, so we treat them as a *fallback* when app metrics are unavailable.

4 Discussion

Principled interaction and safety: Granting an agent raw shell access is brittle and unsafe: the action surface is implicit and shifts over time; arguments lack schemas/units (easy typing/scale errors); permissions collapse to coarse user/group levels; changes are rarely atomic (no clear apply/-commit/revert); failures are non-deterministic with side effects; and logging is ad hoc. A production tuner therefore needs (i) discoverable, typed actions, (ii) argument validation (types, units, ranges, cross-field constraints), (iii) a reversible workflow (commit/revert), (iv) structured, end-to-end audit logs, and (v) fine-grained permission/approval gates. These properties keep an LLM grounded, make unsafe exploration rare and recoverable, and enable post-hoc forensics.

What MCP gives out of the box: The Model Context Protocol (MCP) is a standardized protocol for AI agents to interact with external tools and data sources. It supplies the scaffolding: a JSON-RPC data layer with server primitives—*tools*, *resources*, and *prompts*—discoverable via `*/list`. Tools advertise input schemas via JSON Schema; resources and prompts are discoverable with typed metadata. Utilities include a structured logging channel and change-list notifications for tools/resources/prompts; clients can also offer *elicitation* (user input/approval) and *sampling* (client-mediated model calls). MCP standardizes discovery, schemas, logging, and client UX hooks across heterogeneous servers, reducing bespoke glue and avoiding unsafe shell fallbacks.

What we add on top: While MCP provides the foundational protocol layer, it does not define domain-specific safety mechanisms. To operate safely in an OS setting, we propose layering additional controls above MCP's core protocol (details in Appendix C): (1) a two-phase tuning flow where `set_cfs_params_staged(...)` returns a token and a watchdog enforces guardrails, with the client later calling `commit(token)` or `revert(token)` after measurement; (2) fine-grained permissions via a host/server policy engine that evaluates tool name, caller identity, argument values, and target scope (e.g., CPU/cgroup) with default-deny and per-tool rate limits/cooldowns; (3) approval gates that use elicitation for human confirmation on sensitive actions, with optional auto-approval for low-risk actions; (4) server-side semantic validation beyond JSON Schema to block out-of-band jumps; (5) structured audit trails of calls, decisions, and metric snapshots keyed by run/epoch IDs and idempotency keys (MCP and host logs) to enable replay; and telemetry exposed as MCP resources (`resources/{list,read}`) with typed tools that can attach ephemeral collectors for system metrics.

Cost and latency: LLM inference adds cost, but reduced engineering and faster convergence may often offset it. Unlike BO/RL—which need reward engineering, discretization, and hyperparameter tuning—LLM loops can be refined with lightweight programmatic prompting like DSPy [11]. In our experiments, smaller models (e.g., Gemini 2.5 Flash Lite) exceeded classical tuners—and sometimes larger models—at a fraction of cost/latency (Appendix B.4). Decision latencies are compatible with many OS tuning scenarios; continued improvements are likely to shrink this further [1].

Limitations: Our evaluation validates core capabilities but not a full standalone agent; we decompose the design into smaller, confirmed loops. Results reflect one hardware class, Linux 5.15, and a TPC-C/PostgreSQL workload; broader external validity is open. Objectives emphasize p99 (and IPC), leaving throughput/fairness/energy trade-offs underexplored. BO/RL baselines use reasonable settings, but other discretizations/budgets may narrow gaps; the human baseline is n=1. Short 10 s windows and workload noise can bias estimates; LLM nondeterminism/model drift affects reproducibility. Next steps include lower-latency local models for sub-second loops, MCP-backed tool synthesis from system docs, and multi-agent coordination across CPU/memory/storage/networking.

Future Work: This work introduces a promising paradigm, but also leaves several open directions. (i) API-hosted model latencies constrain sub-second control loops; hybrid designs and smaller local models are a path toward near-instant inference. (ii) MCP governs interactions, but tool implementations still rely on human expertise; future work includes auto-generating and validating tools from sources like manpages to create self-expanding agents. (iii) We focus on a single agent; coordination among multiple agents (e.g., CPU, memory, storage, networking) requires negotiation and conflict resolution to avoid pathological behaviors.

5 Conclusion

This paper presents an LLM-driven, agentic approach to OS tuning. In a CPU-scheduler case study, the LLM loop converges faster and more stably than BO/RL and a human expert—even under dynamic workloads and two-parameter coupling. We sketch a reference architecture with tools for observation, action, and knowledge retrieval; we use MCP for discovery, schemas, logging, and elicitation, and layer host/server permissions, approval gates, and transaction/rollback. The results and design sketch suggest a path toward safe, self-adapting OS control.

References

[1] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. 2025. Small Language Models are the Future of Agentic AI. *arXiv preprint arXiv:2506.02153* (2025).

[2] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2023. Cilantro:{Performance-Aware} resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 623–643.

[3] Huilai Chen, Yuanbo Wen, Limin Cheng, Shouxu Kuang, Yumeng Liu, Weijia Li, Ling Li, Rui Zhang, Xinkai Song, Wei Li, et al. 2024. AutoOS: make your OS more powerful by exploiting large language models. In *Forty-first International Conference on Machine Learning*.

[4] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. `http://www.vldb.org/pvldb/vol7/p277-difallah.pdf`

[5] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. `https://www.flux.utah.edu/paper/duplyakin-atc19`

[6] Johannes Freischuetz, Konstantinos Kanellis, Brian Kroth, and Shivaram Venkataraman. 2025. Tuna: Tuning unstable and noisy cloud applications. In *Proceedings of the Twentieth European Conference on Computer Systems*. 954–973.

[7] Victor Giannakouris and Immanuel Trummer. 2025. λ-Tune: Harnessing Large Language Models for Automated Database System Tuning. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[8] Lao Jiale, Wang Jianping, Chen Wanghu, Wang Yibo, Zhang Yunjia, Tang Mingjie, Li Yufei, Cheng Zhiyuan, and Wang Jianguo. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1939–1952.

[9] Aditya K Kamath and Sujay Yadalam. 2024. Herding llamas: Using llms as an os module. *arXiv preprint arXiv:2401.08908* (2024).

[10] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. 2023. {SelfTune}: Tuning Cluster Managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1097–1114.

[11] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714* (2023).

[12] Brian Kroth, Sergiy Matusevych, Rana Alotaibi, Yiwen Zhu, Anja Gruenheid, and Yuanyuan Tian. 2024. MLOS in Action: Bridging the Gap Between Experimentation and Auto-Tuning in the Cloud. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4269–4272.

[13] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research* 23, 54 (2022), 1–9. `http://jmlr.org/papers/v23/21-0888.html`

[14] Kai Mei, Xi Zhu, Wujiang Xu, Wenyue Hua, Mingyu Jin, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. 2024. Aios: Llm agent operating system. *arXiv preprint arXiv:2403.16971* (2024).

[15] The Linux Kernel Development Community. 2024. *Completely Fair Scheduler (CFS)*. `https://docs.kernel.org/scheduler/sched-design-CFS.html`

[16] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that" Reads the Manual". In *Proceedings of the 2022 international conference on management of data*. 190–203.

[17] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[18] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J Gordon. 2018. A demonstration of the ottertune automatic database management system tuning service. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1910–1913.

[19] Michael Zhang, Nishkrit Desai, Juhan Bae, Jonathan Lorraine, and Jimmy Ba. 2023. Using Large Language Models for Hyperparameter Optimization. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*. `https://openreview.net/forum?id=FUdZ6HEOre`

A Experimental Setup and Implementation Details

This section provides supplementary details on the system configuration, workload, and tuner implementations used in our experiments.

System and Workload Configuration: All experiments were conducted on a dedicated machine with 2× Intel Xeon Gold 6248R @ 3.00 GHz, 192 GB DDR4 RAM, 1 TB NVMe SSD running Ubuntu 22.04 with Linux Kernel 5.15, hosted on Cloudlab [5].

The workload consists of a Benchbase [4] implementation of the TPC-C benchmark running on a PostgreSQL 14 database. The workload was configured with 4 warehouses, and driven by 16 client threads to generate a consistent, high-throughput load on the system. The primary performance objective was the minimization of p99 transaction tail latency, measured in milliseconds at the end of each tuning iteration.

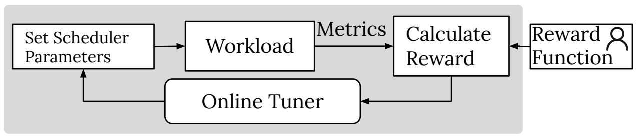

Figure 3: The typical online tuning loop used for all baseline methods. The Online Tuner component is replaced by each of the different strategies (BO, RL, Human, LLM).

Tuner Implementation Details: Our general experimental setup follows the online tuning loop shown in Figure 3. At each step, the tuner proposes a new configuration, which is applied to the live system. The workload runs for a fixed duration of 10 seconds, after which performance metrics are collected and fed back to the tuner.

Experiments overview: We run two experiment types. (i) *Fixed-rate TPC-C:* we evaluate both single-parameter (1P) and dual-parameter (2P) tuning of `min_granularity_ns` and `latency_ns`. (ii) *Variable-rate TPC-C:* a 1P loop on `min_granularity_ns` with a rate decrease from 1100 tx/s to 300 tx/s after tuning cycle 75 which was reverted on tuning cycle 150 (the change points are shown as dashes in the figures). In the variable-rate runs, `latency_ns` is set very low so that `min_granularity_ns` governs timeslices.

Signals and observability: All tuners receive the scalar objective after each cycle. For experiment (ii), the LLM and the human additionally see auxiliary system metrics (e.g., IPC via `perf stat`) and application metrics (e.g., throughput) in the same textual interface (to simulate the on-demand tool calling to gather additional context). Classical baselines use only the scalar reward unless that signal is explicitly redefined (e.g., BO-IPC uses IPC as reward).

CFS Parameter Details: CFS aims to give every task a fair share of CPU time. We tuned: **(a)** `latency_ns`, which sets a target period (in nanoseconds) over which every runnable task should run at least once. The scheduler divides this latency by the number of running tasks to determine an individual timeslice per core. **(b)** `min_granularity_ns`, which enforces a minimum timeslice (in nanoseconds) a task will receive, regardless of the calculation above. It prevents excessive context switching costs on systems with many tasks.

The interaction between these two parameters is non-trivial. If the number of tasks is high, the calculated timeslice from `latency_ns` can fall below `min_granularity_ns`. In this case, the minimum granularity takes precedence, which can cause the total scheduling period to exceed the target latency, impacting application responsiveness. This is a key semantic relationship that traditional tuners often fail to capture.

Parameter Ranges: `latency_ns` $\in$ [2 ms, 100 ms], `min_granularity_ns` $\in$ [0.2 ms, 10 ms] (log-spaced sampling).

Default values on Kernel 5.15 are `latency_ns` $= 24$ ms and `min_granularity_ns` $= 3$ ms.

A.1 Baselines

Bayesian Optimization (BO): As a representative Bayesian optimization method, we adopt SMAC3 [13]. We use its default acquisition function (Expected Improvement), random forest surrogate model, and a fixed exploration-exploitation tradeoff. The optimization is constrained to the same number of cycles as the LLM-based tuning loop to ensure fair budget usage. All runs are seeded (default: 42) to ensure reproducibility. The SMAC3 tuner uses log-scale integer parameter definitions and is configured with `deterministic=False` to account for evaluation noise. Its internal intensifier races configurations aggressively using estimated costs.

Reinforcement Learning: We implement two reinforcement learning baselines: tabular Q-learning and Deep Q-Networks (DQN). Both operate over a discretized parameter grid (8 bins per parameter, logarithmically spaced), yielding a finite action space. The Q-learning tuner uses a 2D Q-table initialized to zero, an ϵ-greedy policy with ϵ decaying from 1.0 to 0.1 at a rate of 0.995, and a fixed learning rate of 0.1. Rewards are scaled for stability and negated for minimization objectives.

The DQN agent is implemented in PyTorch, with a fully connected network consisting of three hidden layers (each with 128 units and ReLU activations), dropout (rate 0.1), and Adam optimizer with a learning rate of 0.001. DQN maintains a replay buffer of size 1000 and trains using batches of 32. Target network updates occur every 10 steps. Exploration follows the same ϵ-decay schedule as tabular Q-learning. The state includes normalized parameter values, reward, iteration number, and time since last change.

Human expert: A PhD-level kernel practitioner with prior experience in scheduler tuning used the same 10 s loop, objective, and auxiliary metrics as the LLM agent baseline. Just like the LLM, the expert had no prior knowledge of the tuned workload or the underlying system hardware, but was only given a brief description of the tunable CFS parameters. No further tuning instructions were provided to the expert, and access to any commands and tools outside the tuning loop that would provide additional hints was restricted. The expert had unlimited time to respond to each new tuning cycle. The expert followed a standard bracket–and–refine routine (coarse doubling/halving then monotone narrowing) and, when rates changed, briefly probed smaller slices to validate sensitivity before returning to the stable region.

A.2 LLM Tuning Loop

Model(s) and latencies: We use `gemini-2.5-flash` via the hosted API. Unless noted, we use a temperature of 0.5, which we found balances early-cycle exploration with stable later-cycle refinement. The LLM operates conversationally, receiving performance metrics (objective and auxiliary) and proposing new parameters in a continuous loop. No external tool calls are made during the loop aside from metric ingestion. We also explore the use of two additional models: `gemini-2.5-pro` and `gemini-2.5-flash-lite`. `gemini-2.5-pro` and `gemini-2.5-flash` use a thinking budget of ten thousand tokens; `gemini-2.5-flash-lite` does not support thinking.

Prompt template: The following is the initial prompt used to guide the LLM agent in the case of the antagonistic knob co-tuning (2P). The conversation then continues with updates on performance metrics and the agent's replies, which include its reasoning and next suggested parameters. For improving the performance of our prompt we relied on the guidelines defined by [19] for LLM-based hyperparameter tuning.

Initial System Prompt for LLM Agent

You are a Linux kernel scheduler tuning expert with deep knowledge of the Completely Fair Scheduler (CFS) and performance optimization. Your goal is to **minimize p99 latency** for a database workload. You have a total of 200 iterations. The workload performance metrics

are NOISY, so do not overreact to single measurements; look for trends. (Optionally: Some auxiliary metrics are provided; you may or may not use them.)
Tunable CFS parameters:

- `min_granularity_ns`: Minimum time slice before preemption. Lower values increase responsiveness but also overhead.
- `latency_ns`: The target latency for all tasks to get a chance to run. Lower values lead to more frequent scheduling.

Parameter Ranges (log-spaced):

- `latency_ns`: 2,000,000 to 100,000,000 nanoseconds
- `min_granularity_ns`: 200,000 to 10,000,000 nanoseconds

Your optimization strategy should be:

1. **Early Cycles:** Prioritize EXPLORATION. Test diverse values across the full range as you see fit.
2. **Later Cycles:** Shift to EXPLOITATION. Refine configurations around promising regions.
3. **Adaptability:** Monitor for extreme changes (e.g., sudden drops or spikes) and increase exploration temporarily if detected.

I will provide performance data after each run. You must provide your analysis and the next configuration to try. Respond ONLY in the format shown below, with no other text:

```
Analysis: <Your one or two-sentence decision reasoning>
Config: { "latency_ns": <int>, "min_granularity_ns": <int>}
```

Conversational Update Prompt Schema for LLM Agent

```
Window: <int>
Config: { "latency_ns": <int>, "min_granularity_ns": <int> }
Reward: { "p_99_latency": <int> }
```

Latest configuration:

```
Window: <int>
Config: { "latency_ns": <int>, "min_granularity_ns": <int> }
Reward: { "p_99_latency": <int> }
Auxiliary Metrics (optional): { "IPC": <float>, "Throughput": <int>, ...}
```

Provide your analysis and the next configuration to try based on the provided format template.

Agent Response Schema

```
Analysis: one or two sentences
Config: { "latency_ns": <int>, "min_granularity_ns": <int> }
```

B Extended Results and Discussion

B.1 Convergence and Stability

We report application p99 latency (ms; lower is better) and improvement over the default configuration (%), broken down by: (i) tuner type (Fixed, Bayesian, RL, LLM variants), (ii) parameter dimensionality (1P vs 2P), and (iii) time windows (All, 0–20, 20–50, 150–200 cycles). The "Params" column denotes the number of tuned knobs; "1 (IPC)" reports p99 when BO optimizes IPC as a proxy.

Discussion:

Table 1: Performance Comparison of All Tuning Methods for different tuning cycle snapshots. All p99 latency values are in milliseconds (ms). $\Delta\%$ shows the percentage difference over the default (Fixed) configuration (< 0 is better.)

		All		0-20		20-50		150-200	
Tuner type	Params	p99	Δ%	p99	Δ%	p99	Δ%	p99	Δ%
Fixed	0	55.85	—	55.90	—	55.78	—	56.22	—
2.5 Flash	1	47.16	-15.55	47.58	-14.88	46.83	-16.04	47.64	-15.27
2.5 Flash	2	53.95	-3.39	55.41	-0.89	52.26	-6.31	54.67	-2.75
2.5 Flash Lite	1	47.67	-14.65	48.65	-12.97	46.50	-16.64	47.77	-15.03
2.5 Flash Lite	2	53.13	-4.87	59.03	5.61	52.06	-6.66	52.51	-6.59
2.5 Pro	1	48.34	-13.44	49.07	-12.22	46.19	-17.20	49.12	-12.63
2.5 Pro	2	55.72	-0.22	56.75	1.52	54.67	-1.99	56.23	0.02
Bayesian	1	49.62	-11.15	49.77	-10.97	49.16	-11.87	49.83	-11.37
Bayesian	2	58.09	4.01	64.71	15.76	60.17	7.87	59.60	6.01
Bayesian	1 (IPC)	49.52	-11.33	49.72	-11.07	48.81	-12.49	49.87	-11.30
DQN	1	49.64	-11.12	49.39	-11.65	48.90	-12.34	49.96	-11.13
DQN	2	62.98	12.78	63.17	13.00	65.90	18.14	60.60	7.80
Q-learning	1	49.17	-11.96	49.71	-11.08	48.85	-12.42	49.21	-12.47
Q-learning	2	61.60	10.30	64.41	15.21	66.05	18.40	57.43	2.16

1P tuning. All methods beat the default, but LLMs lead. Flash attains 47.16 ms (-15.6%), Flash-Lite 47.67 ms (-14.7%), and Pro 48.34 ms (-13.4%), versus BO at 49.62 ms (-11.2%). In absolute terms, Flash improves on BO by 2.46 ms (-5.0% vs BO). Windowed results are consistent: LLMs maintain -13% to -17% across 0-20 / 20-50 / 150-200, showing fast and maintainable convergence.

2P co-tuning of antagonistic knobs. Classical tuners degrade below the default: BO 58.09 ms (+4.0%), DQN 62.98 ms (+12.8%), Q-learning 61.60 ms (+10.3%), while LLMs lead to reduced latency: Flash-Lite 53.13 ms (-4.9%), Flash 53.95 ms (-3.4%), Pro 55.72 ms (-0.2%). Relative to BO, Flash reduces p99 by 4.14 ms (7.1%). Temporal slices show why: early (0-20), BO is 64.71 ms (+15.8% vs default) while Flash is 55.41 ms (-0.9%), a 9.30 ms gap. Mid/late windows keep sizeable margins (7.91 ms / 13.1% and 4.93 ms / 8.3%). Flash-Lite briefly dips early (+5.6%) but recovers strongly (-6.6%, -6.6%).

Proxy objective. BO optimized for IPC (49.52 ms; -11.3%) closely matches BO with a p99 objective (49.62 ms; -11.2%), indicating that in this setup a system-level proxy can preserve application tail latency—supporting the decision to rely on system-level proxy objectives when application metrics are unavailable.

Takeaways. (i) For single-knob tuning, lightweight LLMs (Flash-Lite/Flash) deliver the best p99. (ii) Under antagonistic co-tuning, LLMs retain positive gains while BO/RL regress below default, suggesting that semantic priors help avoid override pathologies. (iii) BO's early-phase instability in 2P explains much of its deficit; LLMs reach good regions faster and hold them. Our observations align with the variable-rate experiment (Fig. 1, right), where the LLM adapts sooner than the human expert while avoiding large missteps.

B.2 Variable Request Rates

We evaluate a variable-rate, single-parameter tuning loop (i.e., `min_granularity_ns`); `latency_ns` is set at 0.1 ms so it is always overridden by `min_granularity_ns`. The request rate steps between two steady states—300 and 1100 tx/s—with a mid-run reduction and subsequent restoration. Table 2 reports application p99 latency (ms; lower is better) and the percent improvement relative to a human expert within each rate window and overall. After the second change point, small continued adjustments reflect 10 s-window noise and conservative trust-region steps; the mean stabilizes while the controller micro-tracks throughput, avoiding large explorations. Post-second shift, the LLM exhibits small fluctuations around the plateau to potentially track mild non-stationarity (DB cache/OS heuristics); tail latency remains stable.

Table 2: Variable-rate tuning (1P). p99 latency (ms) and percent improvement vs human expert within each phase (300 and 1100 tx/s) and overall. The loop tunes `min_granularity_ns` while `latency_ns` is fixed. $\Delta\%$ shows the percentage difference versus the Human tuner (< 0 is better.)

	All		300 tx/s		1100 tx/s	
Tuner type	p99 (ms)	$\Delta\%$	p99 (ms)	$\Delta\%$	p99 (ms)	$\Delta\%$
Human	47.66	—	47.40	—	48.40	—
LLM	46.24	-2.98	45.71	-3.57	47.78	-1.29
Bayesian	49.28	3.40	49.47	4.36	48.72	0.67
DQN	49.58	4.04	49.47	4.37	49.89	3.09
Q-learning	48.88	2.56	49.21	3.81	47.91	-1.01

Discussion: Overall and by phase, the LLM attains the best p99: 46.24 ms overall (-2.98% vs human), 45.71 ms at 300 tx/s (-3.57%), and 47.78 ms at 1100 tx/s (-1.29%). Relative to BO, the LLM reduces p99 by 3.04 ms overall (-6.17%), 3.76 ms at 300 tx/s (-7.60%), and 0.94 ms at 1100 tx/s (-1.93%). Q-learning approaches the LLM at high load (47.91 ms; 0.13 ms higher latency) but lags at the lower rate; BO and DQN trail the human across windows.

Fig. 1 (right) shows why: at each rate change (dashed lines) both the LLM and the human execute a coarse adjustment followed by small refinements and a return toward the prior good setting when the rate is restored, but the LLM reacts earlier (using multiple signals such as throughput and IPC), takes fewer/smaller steps, and exhibits shorter re-stabilization. In contrast, RL methods displayed large, frequent reversals in `min_granularity_ns`—including excursions into unsafe, very small timeslices—which elongate recovery (their trajectories are omitted from the plot for clarity). The gains are largest at 300 tx/s, and remain positive at 1100 tx/s.

B.3 Agent Response Times

We measured end-to-end API latency (including network) over 200 iterations using the same prompt schema and temperature 0.5 as in the main experiments.

Table 3: LLM response latency (seconds) for 200 requests.

LLM model	Thinking	p50 (s)	p95 (s)
Gemini 2.5 Pro	✓	18.63	61.90
Gemini 2.5 Flash	✓	13.76	32.00
Gemini 2.5 Flash Lite	×	1.25	4.52

Discussion: Thinking/reasoning models incur substantially higher latency (e.g., Pro at 18.63 s p50) due to extra compute for multi-step generation and larger outputs. Flash-Lite comfortably fits 10 s control cycles; Flash and Pro are better suited to longer cycles (or tolerating a one-cycle lag) and benefit from tiered use (Lite by default, escalate at change points). Latency scales with prompt/history length; context controls and structured I/O reduce both latency and cost in practice (see Section B.4).

B.4 Agent Cost Analysis

Setup & pricing: All costs use paid-tier list prices (as of August 14, 2025) over 200 iterations: **Pro:** input \$1.25/M, output (incl. thinking) \$10.00/M; **Flash:** input \$0.30/M, output \$2.50/M; **Flash-Lite:** input \$0.10/M, output \$0.40/M. Pro and Flash have a thinking budget of ten thousand tokens per request. In practice, only a fraction of that budget is used.

Discussion: Keeping the full conversation history is usually impractical; capping the conversation window to the last 50 iterations reduces total cost by $\sim$37% for Pro (6.51 to 4.10) and Flash (1.40 to 0.87), and by 56% for Flash-Lite (0.32 to 0.14). In steady-state 10 s loops (8,640 iters/day), this corresponds to approximate daily spend of \$282 to \$177 (Pro), \$60 to \$37 (Flash), and \$14 to \$6 (Lite).[1]

[1]Derived from Table 4; numbers rounded.

Table 4: End-to-end inference cost over 200 iterations.

Model	Input (M)	Output (M)	Total Cost ($)	Cost/Iter ($)
No context cap				
Gemini 2.5 Pro	3.38	0.23	6.51	0.0326
Gemini 2.5 Flash	3.11	0.19	1.40	0.0070
Gemini 2.5 Flash Lite	3.14	0.01	0.32	0.0016
50-iteration context cap				
Gemini 2.5 Pro	1.46	0.23	4.10	0.0205
Gemini 2.5 Flash	1.34	0.19	0.87	0.0043
Gemini 2.5 Flash Lite	1.35	0.01	0.14	0.0007

For further improving cost/latency, several techniques could be used for the OS tuning agent: (i) **Sliding window** to bound input tokens; (ii) **Summarize stale history** into a compact state vector (e.g., current knob values, last k metrics, change rationale); (iii) **Tool-first state** via MCP: store metrics/configs server-side and pass handles/IDs instead of raw text; (iv) **Structured I/O** (JSON-only outputs) to cut verbose generations; (v) **Tiered models**: default to a lightweight model (e.g., Flash-Lite) and escalate to a larger model only on change points or when improvement stalls.

For simple knob tuning with short context, lightweight models are both effective and cost-efficient, and should be preferred. For complex action spaces (multi-subsystem tuning, code synthesis such as eBPF, or long-horizon reasoning), small models with short context may struggle—whether tiered control or fine-tuned local models close this gap remains an open question for future work.

C System Sketch Example: Scheduler MCP Server

We sketch a concrete, MCP-aligned design for a scheduler server that fulfills the safety requirements for a production tuner outlined in the Discussion (discoverable actions, validation, reversibility, audit logs, and permission/approval gates). The agent acts as an MCP *client*; OS, application, and observability backends are MCP *servers*. Servers expose *primitives* (tools/resources/prompts) discoverable via `*/list`; tool inputs use JSON Schema (types/ranges) and we add unit normalization & semantic checks server-side; calls are logged via the MCP logging channel. When a human decision is required, the host can use *elicitation* to obtain approval. Transactional flows, guardrails, and policy/approvals are layered in our server/host; they are not part of MCP itself.

Overview with example: An agent discovers tools via `tools/list`, reads the schema for `os.scheduler/set_cfs_params_staged`, stages `min_granularity_ns=2,000,000`, measures `os.scheduler/metrics/p99_latency`, then commits or reverts—producing structured logs throughout. A server-side watchdog monitors staged changes and can auto-revert if TTL expires or safety thresholds are breached. This replaces the brittle use of the shell with discoverable, schema-checked calls and a reversible workflow.

Typed tools (server-defined): The scheduler server offers tools for controlled CFS updates using a safe two-phase pattern:

(1) Stage change: `os.scheduler/set_cfs_params_staged(latency_ns?, min_granularity_ns, scope?, ttl_sec?, idempotency_key?)` $\rightarrow$ returns `change_token`. Inputs are validated via JSON Schema (required fields; integer types; unit normalization to ns; range checks, e.g., `latency_ns` $\in$ [2e6, 1e8], `min_granularity_ns` $\in$ [2e5, 1e7]). The server enforces *cross-field constraints* (e.g., relationships between `min_granularity_ns` and `latency_ns`). The optional `scope` (e.g., `cpus:[0-3]`, `cgroup:/workload/db`) narrows the target. Long-running actions may emit progress via notifications. Beyond JSON Schema validation, the server could perform semantic checks (e.g., blocking >50% parameter jumps) to prevent pathological configurations.

(2) Finalize change: After measurement (and any required approval via elicitation), the host/agent decides whether to make the staged settings permanent with `os.scheduler/commit_cfs_params(change_token)` or discard them with `os.scheduler/revert_cfs_params(change_token)`.

Example tools/call payload:

```
"jsonrpc": "2.0",
"id": 1,
"method": "tools/call",
"params": {
  "name": "os.scheduler/set_cfs_params_staged",
  "arguments": {
    "min_granularity_ns": 2000000,
    "latency_ns": 24000000,
    "scope": {"cpus": [0,1,2,3]},
    "ttl_sec": 60,
    "idempotency_key": "tune-cycle-42"
  }
}
```

The staged/commit/revert semantics, guardrails, TTLs, and `idempotency_key` are server/host conventions; MCP provides discovery (`tools/list`), invocation (`tools/call`), and schema-based input validation.

Permissions, approvals, and audit (host/server layered): Before tool execution, the host policy engine enforces default-deny, fine-grained authorization over *tool name* (e.g., `os.scheduler/*`), *caller identity/role* (e.g., `agent:llm-tuner`, `role:cfs-writer`), *argument constraints* (e.g., `min_granularity_ns` $\geq$ 1 ms), and *target scope* (CPUs/cgroups). Sensitive actions require an elicitation approval (confirm/deny with optional justification); low-risk actions may be auto-approved with rate limits/cooldowns to prevent oscillation. The server emits structured records (tool, validated args, caller, `change_token`, decision, metrics snapshot, *run/epoch ID*) via the MCP logging channel; the host mirrors these into its own logs for end-to-end traceability and deterministic replay.

Readable resources: To ground decisions without relying on the shell, the server publishes typed, read-only resources, e.g., `os.scheduler/metrics/p99_latency`, `os.scheduler/metrics/ipc`, and current CFS settings. The agent retrieves them via `resources/read` (and, if implemented, server-specific streaming/subscription). Temporary collectors (e.g., PMCs) are managed by tools that start/stop ephemeral collectors (with TTL), surfacing their outputs as resources.

Error handling and recovery: The system provides deterministic failure modes and clear recovery paths: *Schema violations* (JSON Schema validation errors with field/constraint details); *Unknown tool/field* (JSON-RPC "method not found" / validation error); *Authorization denials* (host policy refusal with reason); *Timeouts/crashes* (server-enforced TTL auto-reverts uncommitted `change_tokens`). MCP furnishes the request/response/error envelope; authorization, guardrails, and auto-revert are layered in the server/host.

Composition: Servers are namespaced (e.g., `os.scheduler`, `observability.perf`, `db.postgres`) and compose under MCP's uniform call model. As the scheduler server evolves (e.g., batching or richer transactions), the agent interface remains stable—capabilities are discovered via `*/list` and validated by published schemas; host/server policies continue to govern permissions and approvals outside MCP.

Performance considerations: Tool invocation, schema validation, and logging add negligible overhead relative to multi-second tuning cycles, while the staged commit/revert pattern introduces only temporary state on the server. This governance cost is small compared to the safety and debuggability it provides.

Why fine-grained permissions? OS tuning actions vary in blast radius and reversibility. Argument-aware policy (who may set which knob, by how much, how often, and where—CPUs/cgroups) prevents out-of-band jumps, confines changes to tenants, and throttles oscillation. This reduces incident odds before any LLM decision.

Approval gates vs MCP elicitation: MCP elicitation is a client-side UX primitive that gathers input during a model call. Approval gates are host-enforced governance checks tied to specific tools/arguments (e.g., sub-1 ms granularity on shared CPUs).

D Source Code Availability

We have open-sourced the functional agent prototype at `https://github.com/nebula-cu/An-Expert-In-Residence`