

TuxBot 论文深度解析报告

TuxBot：语义感知的 LLM 在线操作系统调优

——论文深度解析报告

当操作系统有一万多个旋钮，为什么「会读说明书」的大模型比「会算数」的优化器调得更好

论文英文原名：TuxBot: Semantic-Aware Online OS Tuning with LLMs

作者与机构：Georgios Liargkovas、Mihir Nitin Joshi、Kostis Kaffes（Columbia University）；Hubertus Franke（IBM Research）

发表信息：SOSP '26（第 30 届 ACM 操作系统原理研讨会），2026 年 9 月 29 日–10 月 2 日，捷克布拉格；DOI 10.1145/3830418.3843885；CC BY 4.0 开放获取；正文约 20 页

开源实现：github.com/Columbia-DAP-Lab/SemaTune（论文早期曾用名 SemaTune，仓库沿用旧名）

声明：本报告是第三方中文深度解读，不代表论文作者立场。文中所有数字与实验结论均忠于论文原文；凡属笔者的延伸推断，均以「（笔者推断）」显式标注。

0.1 摘要与核心结论先行

如果你只有十分钟，这一章就是整篇论文的全部精华。

问题：现代 Linux 暴露超过 1,200 个运行时旋钮（调度、功耗、内存、I/O、网络），最优配置随负载、硬件和当前运行条件而变，静态默认值注定不是最优。过去十年的答案是自动调优器：Bayesian 优化、强化学习，代表系统是微软开源的 MLOS。但论文用三个活生生的失败案例证明，这类方法有一个共同的、致命的病根——**语义盲 (semantically blind)**：它们把每个旋钮当成一个匿名数值变量去优化一个标量奖励，完全不理解「CPU 最低频率高于最高频率」这种配置为什么荒谬，也不理解「应用指标缺失时 IPC 下降到底意味着什么」。后果是灾难性的：在真实运行的 Memcached 上反复探索出尾延迟恶化 47 倍的配置；在 PostgreSQL 上用错代理指标让 p99 差 2 倍；旋钮从 1 个加到 32 个时 p99 恶化 50%。

方法：论文提出 **TuxBot**——第一个把大语言模型（LLM）放进在线操作系统调优循环的实用系统。它的核心思想是：LLM 的预训练语料里天然包含 man page、内核源码、运维帖子，因此它能像人类专家一样对旋钮的**含义**（而非数值）做推理。为了让这个思路在 live 系统上可用，TuxBot 做了三件事，恰好对应三个工程挑战：(1) **双循环控制（dual-loop）**——便宜快速的 Instant 模型每 1–5 秒做局部探索与纠偏，昂贵的 Reasoning 模型以数十秒为节奏更新整体策略，解决「强模型又慢又贵」；(2) **显式跨 run 记忆**——把历史 session 的轨迹存入向量库，新 session 热启动时检索相似经验，解决「模型不自动积累负载特定经验」；(3) **typed 有界执行（bounded actuation）**——模型只能提议旋钮-取值更新，一切变更经主机侧 Parameter Validator 校验类型、范围与跨旋钮关系后才写入 OS，解决「开放终端接口会自信地闯祸」。

关键数字：在 13 个真实负载、5 个基准套件、最多 41 个 Linux 参数上，在 baselines 不发生灾难性失败的 11 负载子集上，TuxBot 稳定阶段比默认参数提升 **87.2%**，而最强的传统调优器 MLOS 只有 **50.5%**——领先 36.7 个百分点；全部 13 个负载上 TuxBot 提升 **72.5%** 且整个评估过程**零 severe-degradation 事件**，同期 MLOS 反而比默认参数差 31.9%；**只用机器级指标**（应用完全不配合、零改动）也有 **64.0%** 提升，仍然超过拿着应用级指标的 MLOS；Instant 快路径决策延迟稳定在 1.01–2.55 秒（论文称微秒级闭环响应指其调优窗口粒度远快于传统调优器的分钟级节奏，本地控制进程平均只占 0.03 个 CPU 核）；一个 30 窗口的完整调优 session API 成本约 **\$0.20**。

一句话总结：TuxBot 证明了在线 OS 调优的瓶颈从来不是「优化算法不够聪明」，而是「调优器不懂自己在调什么」——把会读说明书的 LLM 关进有护栏的调优循环里，收益、鲁棒性、成本三项同时超过了深耕十年的数值优化路线。

【零基础知识补给站】

本报告是一份「论文深度解析」，假设你没有任何操作系统或大模型背景。文中所有数字（87.2%、50.5%、72.5%、零事故、\$0.20 等）均直接摘自论文原文；论文原文为英文，正文约 14 页加附录，发表于 SOSP——这是操作系统领域与 OSDI 并列的两大顶级会议之一，每两年举办一次，录取率常年在 20% 以下。报告同时参考了论文的开源仓库（Columbia-DAP-Lab/SemaTune，SemaTune 是 TuxBot 在投稿阶段的曾用名，两者指同一系统）。阅读本报告不需要先读原论文——每一章都会从零补齐背景知识，但所有结论都可以回到原文对应章节核对。

Insight#1 这篇论文最重要的身份是「路线判决书」：它没有发明新优化算法，而是用 87.2% 对 50.5% 的对比宣判了「纯数值黑盒优化」路线在 live OS 调优上的终结。

Insight#2 零 severe-degradation 事件是比 87.2% 更值钱的数字——性能可以慢慢调，但把生产服务搞挂一次就足以让运维永久拉黑自动调优（笔者推断）。

Insight#3 \$0.20/session 这个数字的战略意义在于：它把「LLM 进控制循环」从「昂贵的研究玩具」重新定义为「比云实例一小时租金还便宜的运维耗材」。

0.2 如何阅读本报告

本报告按「先讲清问题、再剖析设计、最后评估证据」的顺序展开，章节号与原论文章节大致对应：

- **第 1 章 引言**（对应论文 § 1）：OS 调优为什么难，传统方法谱系（Bayesian 优化、RL、MLOS）为什么不够。
- **第 2 章 失败案例解剖**（对应论文 § 2）：三个失败案例逐一详析——47 倍尾延迟、2 倍代理失真、50% 高维崩溃，以及它们的共同病根。
- **第 3 章 LLM 凭什么、难在哪**（对应论文 § 3）：LLM 为什么能当「语义推理者」，以及把它放进在线循环必须翻越的三座大山。
- **后续章节**：第 4 章 TuxBot 设计（上下文构造、双循环、记忆、校验执行）；第 5 章实现；第 6 章评估（87.2% 等全部数字的出处）；第 7 章相关工作定位；第 8 章讨论与批判性思考；附录章收录参考文献与图表索引。

为了照顾零基础读者，报告中固定出现以下几类版式元素：

- **灰底方框【零基础知识补给站】**：解释正文中用到的前置概念（如什么是 sysctl、什么是 p99 尾延迟），第一次读不懂正文时先读它。
- **酒红色 Insight 段落**：笔者的判断与洞见，是对原文观点的延伸分析，而非复述。
- **琥珀色【注意】框**：容易误读的细节、论文表述与开源实现的出入。
- **图 N**：论文原图（图 1、图 2……），保持原样并附中文导读。
- **示意图 T N**：本报告新增的补充示意图，原论文没有，用于把抽象机制画出来。
- **上标引用 [N]**：指向论文参考文献列表的第 N 条（共 90 条，编号与原文完全一致），第一次出现时会用一句话说明该文献是什么、与本文什么关系。

【零基础知识补给站】

先认识几个贯穿全报告的「主角」。**旋钮 (knob)** 是操作系统暴露给管理员的运行参数，比如 Linux 调度器的时间片长度、CPU 频率的上下限、网络栈的轮询预算——可以把它想象成汽车的中控台：油门响应、空调策略、悬挂软硬都能调，而且边开边调。**LLM (大语言模型)** 是读过海量文本（包括几乎所有公开技术文档）的 AI，它的特长是理解语言的含义，本文利用的正是「它读过内核文档和运维帖子」这一点。**SOSP** (Symposium on Operating Systems Principles) 是系统领域两年一度的最高殿堂，在这里发表论文相当于电影入围戛纳主竞赛（笔者类比）。

Insight#1 本报告的叙事顺序刻意与论文一致：先用三个失败案例「立靶子」，再让 LLM 登场——因为 TuxBot 的每个设计决策都能一一对应回 § 2 的某个失败，跳过失败案例直接看设计会失去理解设计的钥匙。

Insight#2 论文的 90 条参考文献本身就是一部「系统调优小史」：从 2009 年的 iTuned 到 2025 年的 GPTuner，读者可以沿着引用看清这个领域十七年的演化（笔者推断）。

0.3 全局地图：一张图看懂 TuxBot

在深入任何细节之前，请先建立整体图景。下面是论文的 Figure 3，TuxBot 系统总览，把一次调优迭代的完整工作流（图中编号 1-6）画了进去。

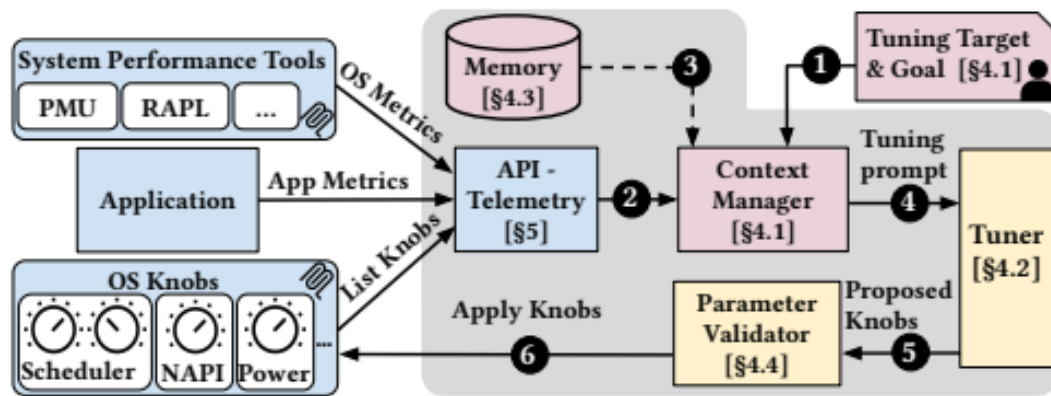


图 3 (论文原图)：TuxBot 系统总览。Session 以类型化的调优请求开始；API-Telemetry 层采集应用与 OS 指标；Context Manager 融合跨 run 记忆构造决策上下文；Tuner 双循环（Instant + Reasoning）产出旋钮提案；Parameter Validator 校验后写入 OS knobs（Power、Scheduler、Memory、NAPI 等）。

怎么读这张图：按编号走一遍。(1) 一次调优 session 从一份「类型化调优请求」开始——声明目标进程、优化目标（如最小化 p99 延迟）与约束（如功耗不超过 60W）。(2) API-Telemetry 层挂接到负载和性能工具（PMU 硬件计数器、RAPL 功耗计等），返回当前旋钮状态和最新测量值。(3) Context Manager 把这些新鲜测量与「跨 run 记忆」（历史 session 的经验）合并。(4) 组装成一份结构化 prompt 发给 Tuner。(5) Tuner 内部是两个循环：快的 Instant 模型负责局部微调，慢的 Reasoning 模型负责战略调整，它们的提案先过 Parameter Validator——这是全系统的「海关」，未知旋钮、类型错误、越界取值、违反声明的跨旋钮关系（如 min_perf_pct 必须不大于 max_perf_pct）一律拦下。(6) 校验通过的更新才写入真正的 OS 旋钮。下一个调优间隔到来，回到 (2)，循环往复。

请先记住这张图的两个「反常识」特征，它们是后文反复回到的主题：其一，**模型永远不碰内核**——LLM 的全部输出只是「提案」，真正的写入发生在主机侧的校验与执行路径里；其二，**图里没有 shell**——TuxBot 刻意不给模型开放式终端，因为作者亲眼见过终端使能的 agent 在实验中跑去关闭 CPU 核、改写块设备调度器（详见第 3 章）。

【零基础知识补给站】

图中的几个英文术语提前混个脸熟。**Telemetry**（遥测）泛指一切从运行中系统采集的测量值：CPU 利用率、每周期指令数（IPC）、内存压力、I/O 等待、功耗——相当于汽车的仪表盘。**PMU**（**Performance Monitoring Unit**）是 CPU 内置的硬件计数器，能精确统计「这个核跑了多少条指令、发生了多少次缓存未命中」，不需要改应用代码。**RAPL** 是 Intel CPU 的功耗计量接口，能读出整个 CPU 封装和内存的实时瓦数。**Prompt** 是发给 LLM 的输入文本，TuxBot 的 prompt 是高度结构化的：角色、目标、旋钮说明书、当前配置、最新测量、近期动作历史，像一张填好的病历。**双循环**里的 Instant 和 Reasoning 都是 LLM，区别只在模型档位：前者用便宜快速的小模型（gemini-2.5-flash-lite），后者用推理能力强的大模型（gemini-2.5-flash）。

Insight#1 整张架构图里最耐人寻味的组件是 Parameter Validator：它把「LLM 的能力」与「LLM 的权限」彻底解耦——能力可以很大（理解全部文档），权限必须很小（只能提议白名单内的旋钮）。这个「大能力、小权限」原则可能是 LLM 进生产系统的通用范式（笔者推断）。

Insight#2 注意数据流是闭环的：写入旋钮后下一个窗口的测量会回流到上下文——TuxBot 本质是一个控制系统，LLM 只是其中的控制器，这决定了它必须回答经典控制论问题：延迟、稳定性、振荡（第 6 章的 Table 3 正是回答这个问题）。

Insight#3 图 3 中没有画出「失败回路」：校验失败的提案会被记录进下一条 trace 喂回给模型——模型连「我刚才为什么被拒」都能看到，这是它区别于传统调优器的隐性行为学习能力（笔者推断）。

第 1 章 引言：为什么操作系统调优是个「读不懂说明书就会翻车」的问题

对应原论文 § 1 (Introduction)。本章回答三个层层递进的问题：操作系统为什么需要「边跑边调」？过去十年的自动调优方法为什么始终差一口气？论文凭什么认为 LLM 是那把缺失的钥匙？

1.1 一万个旋钮与一个不可能的任务

现代操作系统是一个巨大的控制面。论文开宗明义：现代 OS 暴露了**大量运行时控制**——CPU 调度器（决定哪个线程下一个跑、能跑多久）、功耗管理（决定 CPU 跑多快、闲时睡多沉）、内存子系统（决定何时回收页、何时回写脏数据）、I/O 与网络栈（决定中断怎么合并、轮询多久）。光 Linux 内核暴露的可调旋钮就超过 **1,200 个**，而且还有继续增加的提案^[15]（文献 [15] 是 Xkernel, OSDI '26 上的一项工作，系统性地研究如何把内核里写死的性能常量变成安全的运行时旋钮；本文引用它佐证「旋钮数量还在膨胀」这一事实，后文 § 2.3 还会用到它）。

问题的关键不在旋钮多，而在**没有一组旋钮值对所有情况都对**。论文引用了一系列工作^[2-4, 10, 15, 26, 44, 45]说明：最优配置依赖三件事——负载（Memcached 这类延迟敏感服务和批处理任务想要完全相反的设置）、硬件（在这颗 CPU 上安全的频率范围换台机器可能就不安全）、以及**当前运行条件**（同一个服务，上午低峰和晚高峰的最优配置不同）。这就排除了「出厂调好一次」的可能性：Linux 发行版的默认值是面向「不出事」的保守折中，而非面向「你的负载最快」的最优解。默认值与最优值之间的差值，就是调优要收割的性能余量。

于是「谁来调」成了核心问题。人类专家当然能调——他们读 man page、看监控面板、凭经验改参数、观察效果再迭代。但论文指出这条路**不可扩展**：懂内核调度器和 DVFS 交互的专家是稀缺资源，而一个数据中心里有成千上万种负载、成千上万台机器，专家调优的速度永远追不上负载变化的速度。剩下的路只有一条：把调优做成一个**在线闭环**——观察、决策、调整、再观察，机器自动完成。

【零基础知识补给站】

什么是「调优」？用空调打比方。想象一栋大楼的中央空调系统：有上百个阀门和风速档位（旋钮），每个房间的 occupancy（占用率，类比负载）全天都在变。物业给所有阀门设了一个「通用中档」（默认值）——不出错，但电费高、有人喊冷有人喊热。「调优」就是根据每个房间的实时情况动态调阀门，让舒适度（性能）最高、电费（功耗）最低。人工调是一个老师傅拿着对讲机满楼跑（不可扩展）；自动调优是装一套控制系统替老师傅跑腿。而本文的核心冲突是：过去的控制系统只**看得懂温度计的读数**（标量数值），却看不懂贴在阀门上的说明书（「此阀门与三楼新风阀联动，不可单独开最大」）——于是它会把两个「各自合法、组合荒谬」的档位同时打开。TuxBot 的思路，就是换一个读得懂说明书的控制员。

Insight#1 论文把「默认值保守」作为全部工作的前提——这其实是一个可证伪的市场判断：如果默认值已经离最优很近，调优就不值得做。87.2% 的稳定期提升说明这个余量不仅存在，而且大得惊人。

Insight#2 「负载 × 硬件 × 运行条件」三元依赖是调优问题的一切复杂性来源：它意味着不存在可离线计算的答案，任何方案都必须在 live 系统上边测边学——这正是后文所有风险（搞挂生产服务）的来源。

1.2 界定研究对象：稳态 out-of-band 调优

「在线调优」这个词很容易让人误以为论文要替换 CPU 调度器本身。论文用 Figure 1 精确地划定了自己的地盘。

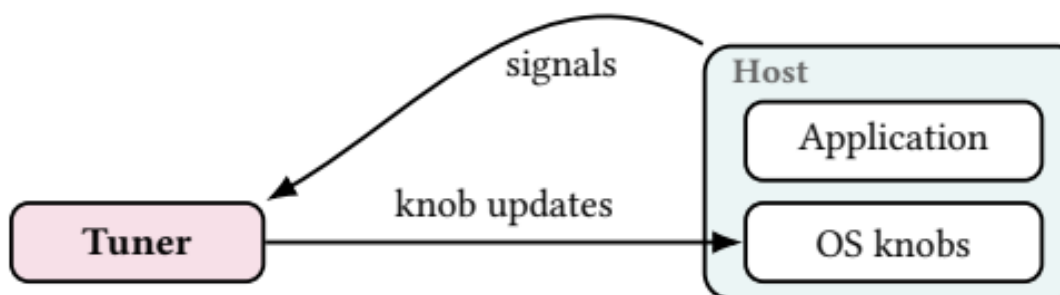


图 1（论文原图）：稳态在线调优循环。左侧是主机侧调优器（host-side tuner），右侧是运行中的主机（Application 跑在 OS 之上，OS knobs 暴露控制面）；调优器周期性地读取信号、写入旋钮更新，不在任何请求的关键路径上。

怎么读这张图：图中央是一台正在跑应用的主机，应用下面是 OS，OS 侧面伸出一排「旋钮」。调优器站在主机旁边（而不是应用和 OS 之间），按自己的节奏做三件事：读信号（遥测）→ 决定新配置 → 写入旋钮。注意箭头不经过应用——应用照常收请求、响应，完全不知道旁边有个调优器在拧旋钮。

论文用一组对照把研究对象钉死。它**不是**内核快路径控制器：CPU 调度器本身、包调度器、TCP 拥塞控制这些组件工作在微秒到毫秒级，每个请求、每个数据包都要经过它们，那是另一片研究战场^[14, 28, 33, 52]（这组文献是「用 ML 增强内核快路径」方向的代表：文献 [14] 研究 Linux 内核负载均衡的 ML 方法，[28] 是 ASPLOS 2023 的 LAKU 框架，[33] 是 OSDI 2020 上用轻量神经网络预测闪存延迟的 LinnOS，[52] 是 EuroSys 2025 的存储 I/O 准入控制工作 Heimdall；本文引用它们只为划清界限，不与它们竞争）。TuxBot 所在的位置叫 **out-of-band（带外）稳态调优**：在**秒到分钟**的时间尺度上，调节那些快路径控制器的**参数**——比如调度时间片的长度、网络栈的轮询预算——让快路径控制器在一个更好的设定点上工作。

这个定位带来两个重要推论。第一，TuxBot 的决策不必快到微秒级：它有整整一个测量窗口（论文用 5 秒）来思考和行动，这正是「LLM 这种秒级响应的组件能被放进循环」的前提。第二，它的目标负载是**稳态、长寿命的服务**——需求持续存在、值得花时间找到好配置并长期享受收益；短任务和毫秒级爆发的过载场景不在其列（论文 § 4.5 也坦承了这条边界）。

【零基础知识补给站】

「带内」与「带外」、快路径与慢路径。还以大楼空调为例：温控器根据室温每秒调整压缩机启停

，这是**快路径**——它在「制冷」这个动作的关键路径上，慢一毫秒都不行。而物业工程师每周根据天气预报和电费账单调整「温控器的目标温度曲线」，这是**带外慢路径**——它不改变制冷机制本身，只改变机制的工作设定点。对应到 OS：CPU 调度器每个时钟滴答都在决定谁上核（快路径，纳秒-微秒级）；而调度器的「最小时间片」参数（`sched_min_granularity_ns`）决定了这个决定的大致粒度，调这个参数就是带外调优。为什么这个区分生死攸关？因为快路径组件换错的代价是系统立刻抖动甚至崩溃，业界对它们的改动极其保守；而带外调优「只是改参数」，出错可以回退——这给「用一个偶尔会犯错的 LLM」留出了安全空间（笔者推断：这是论文选择这个生态位的深层原因）。

Insight#1 「秒到分钟时间尺度」这个定位是整个系统的「合法性来源」：它既解释了为什么 LLM 来得及思考，也解释了为什么论文不追求亚秒级过载响应——不是做不到，而是那本来就是另一个问题。

Insight#2 Figure 1 里调优器与主机并排而立而非嵌入内核，这个「外置」形态与 MLOS 等前辈一脉相承——TuxBot 的创新不在调优器的位置，而在调优器的「大脑」（笔者推断）。

1.3 传统方法谱系：从 Bayesian 优化到 MLOS

自动调优不是新课题。论文在 § 1 扼要回顾了主流路线：人工专家之外，自动化的两大支柱是 **Bayesian 优化**和**强化学习 (RL)** ^[31, 71, 80]（文献 [31] 是 Google Vizier，工业界最著名的黑盒优化服务；[71] 是 NeurIPS 2012 的经典论文，奠定了用 Bayesian 优化调机器学习超参数的范式；[80] 是 1992 年提出 Q-learning 的 RL 奠基文献。三篇合起来就是传统调优器的「方法论基因库」）。在这条基因线上长出了一批代表系统，论文点名了六个：

- **CherryPick**^[5]（NSDI 2017）：为云上的大数据分析作业（如 Spark）自动选择最佳云配置，用 Bayesian 优化在「又贵又慢的搜索」中做到足够少的采样。它是「调优即搜索」范式的早期代表。
- **SmartConf**^[78]（ASPLOS 2018）：自动识别并调整性能敏感的配置项，强调「先找出哪些参数真的重要」再调。
- **OPPerTune**^[72]（NSDI 2024）：微软的部署后服务调优系统，用 RL 同时处理数值型和类别型旋钮——它已经是「在生产服务上线调」的成熟形态。
- **TUNA**^[29]（EuroSys 2025）：针对不稳定、噪声大的云应用调优，引入多保真度采样、不稳定配置过滤和噪声感知调整——它已经在修补「数值优化的脆弱性」，但补丁本身仍是数值层面的。
- **MLOS**^[20, 51]：微软开源的自动化性能工程基础设施（文献 [20] 是 DEEM 2020 的架构论文，[51] 是 PVLDB 2024 的实战报告），把「可调参数」和「度量」外置成标准接口，核心用 SMAC3 做 Bayesian 优化。它是本论文评估中**最强的 baseline**，也是第 2 章失败案例的「主演」。
- **SelfTune**^[47]（NSDI 2023）：在线调集群管理器（如 Kubernetes 调度）的参数，代表了「调优对象从单机服务扩展到集群系统」的一支。

这些系统方法论各异，但共享同一个世界观：**每个旋钮是一个数值变量，目标是把一个标量奖励函数优化到极值**。旋钮的名字、文档、所属子系统、与其他旋钮的关系，在优化器眼里统统不存在——min_perf_pct 只是一个取值范围 0-100 的数，它和 max_perf_pct 之间「最小不得超过最大」的关系，不硬编码进约束就无人知晓。论文给这个病根起了名字：**语义盲**。下一章将看到，语义盲不是理论洁癖，而是会在生产服务上炸出 47 倍尾延迟的真实事故。

【零基础知识补给站】

Bayesian 优化与强化学习到底是什么？ 假设你要调一台收音机找最清晰的电台，但你蒙着眼，每次只能转一下旋钮、听一秒噪音大小。**Bayesian 优化**像一个会记笔记的听众：它记住「上次转到 92.5 噪音小」，在脑内建一个「噪音随频率变化的概率地图」，下一步优先去地图上说「这里可能清晰、或者我完全没探过」的位置。它样本效率高，但地图是纯数值的——它不知道 92.5 是「交通广播频段」还是「军用电台频段」。**强化学习**像一个训练宠物的过程：动作（调旋钮）之后给奖励（性能变好 +1，变差 -1），算法逐渐学会「在什么状态下做什么动作」。它天生处理序列决策，但需要海量试错，而每次「错」都可能是生产服务的一次抖动。两者的共同盲区：状态、动作、奖励全是数字。说明书、旋钮名、子系统结构这些「文字信息」对它们而言不存在。这正是 LLM 能补的位——LLM 的全部本领就是处理文字的含义。

Insight#1 论文选择 MLOS 作为头号对手是聪明的：它是工业界最强、维护最好、唯一能扩展到大型混合旋钮空间的传统系统——打败它就约等于打败整条路线（笔者推断）。

Insight#2 六个系统横跨 2017-2025 年，说明这不是「老古董 vs 新技术」的对比，而是「当前 SOTA vs 新范式」的对比——传统路线一直在进步（TUNA 的噪声过滤就是证据），但论文要证明的是它们在错误的维度上进步。

Insight#3 「语义盲」这个命名是全文最重要的概念武器：它把分散的失败（配置矛盾、代理失真、高维崩溃）统一归因，从而把「修 bug」级别的改进诉求升级成「换范式」级别的论证（笔者推断）。

1.4 论文的核心洞察、三个挑战与贡献清单

在回顾完传统路线后，论文 § 1 抛出了全篇的核心洞察：**LLM 能对旋钮与遥测的「含义」做推理**。LLM 的预训练语料天然包含 man page、内核源码、运维帖子，因此它能像人类专家一样在上下文中解读候选配置——拒绝语义不健全的组合（既包括 minperf_pct > maxperf_pct 这类直接冲突，也包括数值合法但对当前负载无意义的组合）、提出修复建议、并且**主动避开人类运维根本不会去探索的区域**。更妙的是，当应用指标（如服务的 p99）缺失时，模型可以解读**联合遥测签名**——CPU 饱和、运行队列增长、内存压力、功耗状态行为、I/O 等待的组合模式^[18, 40, 61, 70, 81]（这组文献是「用硬件计数器与系统信号推断应用状态」方向的经典：[18] 是 OSDI 2004 年关于关联插桩数据与系统状态的开创性工作，[40] 是 SC 2012 年测量数据中心应用间干扰的工作，[70] 是 ASPLOS 2008 年的请求签名工作，[61] 与 [81] 是近年 OSDI 上的内存分层管理研究；本文用它们说明「多信号联合推断」在学术上有深厚根基，而 LLM 把这个需要专门建模的能力变成了「开箱即用的常识」），无需任何训练即可推断负载是否正走向更好的工作点。

但把 LLM 放进在线调优循环绝非「接上 API 就行」。论文诚实地列出了三个工程挑战：(1) **成本与延迟**——最会推理的模型又慢又贵，而它思考的时候服务还在跑，控制决策等不起；(2) **无记忆**

——预训练模型有通用系统知识，但不自动积累负载特定经验，每个 session 都从零开始；(3) **开放终端的危险**——给模型一个 shell，它就可能发出任意命令、组合出有害操作、改动无关的主机状态。对应的三个解法——双循环控制器、显式记忆、typed 有界执行——将在第 3、4 章详细展开。

§ 1 末尾还给出了评估的「预告片」，这些数字值得提前记住：13 个负载、5 个基准套件、最多 41 个参数；在 baselines 不发生灾难性失败的 11 负载子集上，TuxBot 稳定阶段提升 **87.2%**，MLOS 为 **50.5%**；仅用系统级指标时 TuxBot 仍有 **64.0%**，反超有应用反馈的 MLOS 达 13.5 个百分点；全部 13 负载上 TuxBot 提升 **72.5%** 且 **零 severe-degradation 事件**，MLOS 则比默认参数还差 31.9%。被剔除出子集的两个负载（Xapian 和 Memcached）上，多个 baseline 陷入了论文所称的**队列主导的亚稳态（metastable）退化**^[9, 36]（文献 [9] 是 HotOS 2021 年首次命名「亚稳态失效」的论文，[36] 是 OSDI 2022 年在真实系统中观察这类失效的后续研究；它们指出现象——系统在过载后即使撤掉诱因也无法自愈——正是 TuxBot 完全避开、而传统调优器反复踩中的陷阱）。

最后，论文的四条贡献声明原文照录：(1) 展示现有在线 OS 调优器因缺乏语义理解而在 live 系统上失败；(2) 提出 TuxBot——首个结合 cost-aware 双循环控制、显式记忆、typed 有界执行的 LLM 在线 OS 调优器，即使应用级指标不可用也能有效调 Linux 旋钮；(3) 展示 TuxBot 平均超过 SOTA 调优器 36.7 个百分点（应用指标）/ 13.5 个百分点（仅系统指标），且在评估运行中零 severe-degradation 事件。

【零基础知识补给站】

什么是「亚稳态」（metastable）？ 想象一个拥挤的地铁站：高峰人潮涌入，闸机前排起长队；即使高峰过后、进站人数恢复正常，队却散不掉——因为队伍本身造成了二次拥堵（乘客绕行、闸机利用率下降）。系统卡在了一个「坏的稳定状态」：诱因已消失，伤害却自我维持。计算机系统里有完全同构的现象：服务一旦队列积压，重试、超时、缓存失效会制造额外负载，让系统永远追不上积压——论文的 Xapian 案例里，MLOS 把系统推进这种状态后，稳定期 p99 高达 80 秒（是默认配置下性能的 1/1600 左右，相对提升 -99.9%），且移除恶配置后仍不恢复。这就是为什么调优器的安全标准不能只是「别选坏配置」，而必须是「别把系统推入撤掉坏配置也回不来的区域」。

Insight#1 三条贡献与三个挑战、三个失败案例形成工整的「3×3 对应」——这种结构对称性说明论文的问题定义与解法设计是一起成型的，而非先有系统再凑故事（笔者推断）。

Insight#2 注意贡献 (2) 的措辞「即使应用级指标不可用也能有效调」：这是全文最具产业冲击力的卖点，因为现实中绝大多数遗留服务根本导不出干净的 p99 指标——能用机器指标调好它们，等于把自动调优的适用面扩大了一个数量级（笔者推断）。

Insight#3 「零 severe-degradation」被写进贡献声明而非只放在评估里，说明作者清楚：对运维而言，调优器的第一美德不是聪明，是不闯祸。

第 2 章 失败案例解剖：在线 OS 调优为什么是「语义盲」的

对应原论文 § 2 (Why Online OS Tuning Is Semantically Blind)。这是全文最短却最重要的一章：它用三个真实发生在 live 系统上的失败案例，为整篇论文「立靶子」。论文选择 MLOS 作为贯穿案例——理由是充分的：MLOS 是评估中最强的 baseline，且干净地实例化了经典「采样-建模-推荐」调参循环；它的失败不是实现 bug，而是范式的失败。三个案例分别对应三种失败形态：**数值合法但策略荒谬、奖励缺失时代理误导、旋钮增多时风险爆炸**。

2.1 案例一：数值合法 ≠ 好策略（Memcached 47 倍尾延迟事件）

第一个案例的论点是一句话：**MLOS 把每个旋钮当独立数值变量，但内核把调度、CPU 功耗、内存、I/O 设置组合成「一个」运行时策略**。每个旋钮单独合法的配置，组合起来可能是语义不健全的。论文引用配置分析工作 Diffy^[38]（文献 [38] 是 PLDI 2024 上用数据驱动方法从配置中找 bug 的研究；它的结论——「良构（well-formed）的配置仍然是故障的常见来源」——恰好为本文站台：这个问题在静态配置世界已被证实，在线调优把它进一步放大了）。

「语义不健全」在论文里涵盖两类：(a) **内部矛盾的策略**（semantically violating），例如把 CPU 最低频率百分比设得高于最高频率百分比；(b) **数值合法但对负载无意义**（nonsensical），例如给一个延迟敏感服务同时开极端 busy polling、浅 idle state 和超长调度窗口——每个值都在合法范围内，组合起来却像「一边猛踩油门一边拉手刹」。

下面是论文 Table 1 的完整重建。场景：Memcached 在 500,000 QPS 高负载下运行，MLOS 协同调 8 个 OS 旋钮，目标最小化 p99 尾延迟。表中记录了约 4 分钟内 6 个代表性时刻：

时间 (s)	p99 (ms)	吞吐 (req/s)	配置类别	系统状态 / 配置说明
10	1.43	499,995.1	Valid (正常)	Stable state: 尾延迟稳定在健康水位
20	64.18	458,990.6	Unsound (无意义)	busypoll=975; 浅 C-state
25	68.38	466,478.1	Unsound (无意义)	busypoll=702; 50 ms 调度时间片
150	1.78	499,389.8	Repaired (已修复)	Tail recovers: 尾延迟恢复
225	15.02	498,136.4	Unsound (违例)	minperfct > maxperfct (70% > 10%)
250	54.44	425,471.4	Unsound (违例)	minperfct > maxperfct (63% > 3%) ; busypoll=617

表 1（重建自论文 Table 1）：Memcached 500k QPS 下 MLOS 的调优轨迹。「Unsound」指语义不健全配置——注意它们在 4 分钟内反复出现。

逐行导读这条轨迹。前 10 秒一切正常：p99 只有 1.43 ms，吞吐几乎打满 50 万 QPS。第 20–25 秒，MLOS 探索到「数值合法但无意义」区域：极端的 busy polling（网络栈空转轮询 975/702 微秒）叠加浅 idle state 和 50 毫秒级的超长调度时间片——这对一个微秒级响应的服务是灾难配

方，p99 冲到 64–68 ms（**相对健康水位恶化超过 47 倍**），吞吐跌至 459–466k。到第 150 秒，MLOS 远离了坏配置，尾延迟恢复到 1.78 ms。但故事没有结束：第 225 秒，MLOS 设置了 $\text{minperfct} > \text{maxperfct}$ ($70\% > 10\%$) ——一个直接自相矛盾的功耗策略，此时吞吐仍接近目标 (498k)，p99 却已悄悄涨到 15.0 ms；第 250 秒，又一次 $\text{minperfct} > \text{maxperfct}$ ($63\% > 3\%$) 叠加激进 busy polling，p99 回到 54.4 ms，吞吐跌至 425k。

这条轨迹包含三个值得刻进脑子的事实。第一，语义盲的探索**不是踩一次坑，而是反复踩两类坑**——4 分钟内两次违例、两次无意义组合，因为优化器无法从「上次为什么惨」中学到语义层面的教训。第二，**接近正常的吞吐会掩盖严重的尾延迟损害**：225 秒时吞吐 498k（几乎满分），p99 却已恶化 10 倍——只看聚合吞吐的监控会完全错过这类伤害。第三，论文把这类「持续多窗口的严重退化」定义为 **severe degradation incidents**，并在 § 6.5 给出了形式化判据（至少连续 3 个窗口、即 15 秒，性能差于默认值的 10 倍），后面评估章节会用这把尺子量所有调优器。

有人可能会问：手动把「 minperfct 不得大于 maxperfct 」这类约束加进优化器不就行了？论文预先堵住了这条路：**交互数量太多，且坏配置依硬件与负载而定**。一对旋钮的约束好写，41 个旋钮两两组合就有 820 对，更何况「对延迟敏感服务无意义」这种约束根本无法写成静态不等式——它依赖对负载语义的理解。

【零基础知识补给站】

表中的术语逐个解释。Memcached 是最流行的内存缓存系统，互联网公司的「热点数据兜底」层，对延迟极度敏感——它的 p99 从 1.4 ms 涨到 64 ms，意味着每 100 个用户请求里就有一个要多等 60 多毫秒，足以拖垮上游网页加载。**p99 尾延迟**：把所有请求的响应时间从小到大排队，第 99% 位置的那个值——它衡量「最倒霉的 1% 用户」的体验，比平均值更能暴露偶发卡顿。**busy polling（忙轮询）**：网络包到来时 CPU 不睡觉、空转等待下一个包，用耗电换取微秒级的响应——对低负载有用，高负载下纯属浪费核。**C-state** 是 CPU 的「睡眠深度」等级，C1 最浅、越深越省电但醒来越慢；「浅 C-state」就是不许 CPU 睡沉。**minperfct / maxperfct** 是 Intel CPU 频率的下限与上限（百分比）；下限 $70\% >$ 上限 10% 就像规定「车速不得低于 70 km/h 且不得高于 10 km/h」——内核拿到这种指令只能做未定义行为式的取舍（笔者推断）。

Insight#1 47 倍这个数字的真正杀伤力在于它发生在「调优器正常工作」的过程中——MLOS 没有 bug，它严格按设计探索了搜索空间；把生产服务当探索成本，正是数值范式的原罪。

Insight#2 「吞吐正常 + p99 爆炸」的组合揭示了一个监控盲区：调优器伤害的往往是最脆弱的那 1% 请求，而聚合指标看起来岁月静好——这解释了为什么这类事故在真实运维中常常事后才发现（笔者推断）。

Insight#3 两次 $\text{minperfct} > \text{maxperfct}$ 违例说明 MLOS 的 SMAC3 内核把这对旋钮当独立变量采样；约束不是不能加，而是「加约束」这个思路在 41 旋钮的规模下必然漏成筛子——语义理解才是可扩展的约束来源。

2.2 案例二：奖励缺失，代理失真（PostgreSQL 的 2 倍差距）

第二个案例把问题从「动作侧」移到「奖励侧」。理想的调优循环需要应用直接报告性能指标（服务的 p99、吞吐），但论文指出一个现实：有些服务能暴露稳定指标，有些则藏在专有栈、遗留代码或「运维不想频繁采样」的插桩后面。一个实用的调优器必须在两种场景下都能工作。

应用指标缺席时，传统调优器的标准做法是找一个低层「代理指标」顶上——最常用的是 IPC（每周期指令数）和 cache misses（缓存未命中数）。直觉上看它们很合理：IPC 高说明 CPU 干活效率高，cache miss 少说明数据局部性好。但论文用 Figure 2（左）给出了反证：PostgreSQL 跑 Wikipedia 负载^[21]（文献 [21] 是 OLTP-Bench/BenchBase，数据库基准测试的标准工具，Wikipedia 是其中的一个真实感负载；本文的多个实验都跑在它上面），让 MLOS 分别优化三种标量目标。结果：用应用指标（App）时 p99 相对默认配置下降；改用 IPC 或 Cache misses 作为目标时，p99 明显更差——按论文引言的量化口径，**约为用应用指标调优的 2 倍差**。

为什么会这样？论文的诊断一针见血：问题不在遥测缺失，而在**没有任何单一低层信号能跨负载、跨工作点可靠地代理应用性能**。同样的 IPC 值，在不同调度行为、内存压力、I/O 活动下含义完全不同：高 IPC 可能是「高效处理请求」，也可能是「自旋锁空转」；cache miss 上升可能是「工作集变大」，也可能是「预取器被搞乱」。低层信号是「事实」，但从事实到「应用体验变好还是变坏」的翻译，本身就是语义问题。

论文还顺手否定了两条常见的修补路线。其一，把多个机器指标加权组合成奖励^[70]——这只是把「选哪个信号」的问题转移成「选什么权重」的问题，而正确权重同样依赖负载和工作点。其二，内存分层系统的研究已经证明单一信号（如页面热度）不够、需要多信号策略推理^[22, 61, 77, 81]（文献 [22] 是 HPDC 2019 的 Kleio 混合内存页调度器，[61] 与 [77] 分别是 OSDI 2025 和 SOSP 2024 上超越「热度」单信号的内存分层管理研究，[81] 是 OSDI 2024 的 Nomad；本文借它们说明：即使在研究最充分的子领域，「多信号联合推理」也是公认刚需）。

【零基础知识补给站】

什么是「代理指标」？为什么它会骗人？ 医生想知道你的心脏健康程度，最直接的是心脏彩超（应用指标），但彩超贵且慢；日常体检用心率（代理指标）代替。大多数时候心率能反映心脏状况——但你刚爬了楼梯、刚喝了咖啡、正在紧张，心率都会偏高，与心脏健康无关。IPC 之于应用性能就是心率之于心脏：相关，但受太多无关因素调制。更糟的是，调优器是「按指标发奖金」的制度：你奖励 IPC，系统就会往「IPC 高但应用更慢」的方向演化——这是古德哈特定律（「指标一旦成为目标，便不再是好指标」）在系统调优中的经典上演（笔者类比）。TuxBot 的破法是后文第 3 章的主题：不找单一代理，而是让 LLM 同时阅读十几路信号，像老中医「望闻问切」一样综合判断。

Insight#1 这个案例的深层含义是：「应用指标可得性」不应成为调优器能力的前置条件——把「读懂系统信号」内化为调优器的智能，比要求每个应用改造上报接口，工程上现实得多（笔者推断）。

Insight#2 2 倍差距发生在 PostgreSQL + Wikipedia 这种「最标准」的数据库场景上，说明代理失真不是边缘案例而是主流现实；连教科书场景都代理不住，长尾服务更不必说。

Insight#3 加权组合路线被否定的理由值得玩味——「正确权重依赖负载与工作点」意味着任何固定标量化都注定局部最优；要动态地解释信号组合，就需要某种「理解」，而不仅是「计算」。

2.3 案例三：旋钮越多，语义风险越大（1→32 旋钮 p99 恶化 50%）

第三个案例处理「规模」维度。直觉上，给调优器更多旋钮应该给性能更多上升空间；现实是，对传统调优器而言更多旋钮首先意味着更多闯祸方式。论文回忆了两个背景事实：Linux 暴露 1,200+ 可调旋钮且仍在增加^[15]；自动化 OS 专业化研究早已发现，大配置空间里含有大量无效乃至致故障的区域^[1, 66]（文献 [1] 是 Inria 2019 年关于学习超大 Linux 配置空间的技术报告，[66] 是 JSS 2021 年关于学习软件配置空间的系统性文献综述；两者共同支撑「大空间 = 大量危险区域」的判断）。

机制上有两层放大。其一，加旋钮不只扩大搜索空间（维度诅咒），还**倍增跨旋钮交互**——8 个旋钮有 28 对潜在交互，41 个就有 820 对。其二，每次「错误的更新」不只是浪费一次采样，还可能把主机推入**难以恢复的状态**：队列积压、缓存破坏、写回压力——这些状态有惯性，撤掉配置后伤害仍在持续。

Figure 2（右）给出了量化证据：PostgreSQL 跑 TPC-C（最经典的数据库事务负载），MLOS 在 1-2 个旋钮时还能待在默认配置附近；当调优面从 1 个扩展到 32 个旋钮，p99 急剧恶化——按论文引言的量化口径，**退化 50%**。也就是说，「更大的控制面」在数值优化器手中变成了净负资产。

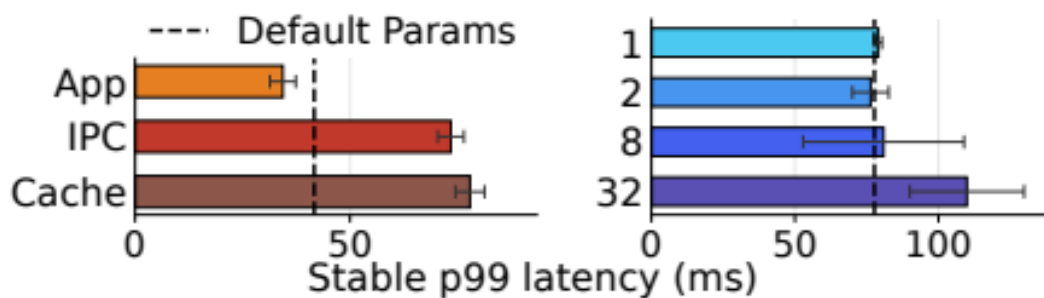


图 2（论文原图）：MLOS 调优失败的两个侧面。左：PostgreSQL 跑 Wikipedia，MLOS 分别优化应用指标（App）、IPC、Cache misses 三种目标时的稳定 p99——两种低层代理明显更差；右：PostgreSQL 跑 TPC-C，调优面从 1 扩到 32 个旋钮时稳定 p99 的恶化曲线。

怎么读这张图：左图三根柱子是同一负载、同一调优器，唯一变量是「奖励信号选什么」——柱越高越差。用 App 时 MLOS 能改进默认值；换成 IPC 或 Cache misses，p99 反而比默认更糟（论文引言口径：约为用应用指标的 2 倍差）。这等于说：同一台优化机器，给它错误的「意义系统」，它会把改进变成伤害。右图的横轴是参与调优的旋钮个数（1、2、8、32），纵轴是稳定 p99：1-2 个旋钮时 MLOS 还贴着默认区域，旋钮一多曲线就抬头向上（论文引言口径：1→32 旋钮 p99 退化 50%）。两幅子图合读就是 § 2 的核心信息：**传统调优器的能力被「语义可得性」和「控制面规模」两头挤压。**

【注意】

论文 Figure 2 的左、右两幅子图在文本提取中只能看到坐标轴刻度（左图纵轴 "0 50"，右图 "0 50 100"），各柱/各点的精确数值原文正文未给出。本报告对这两幅图一律采用论文 § 1 引言给出的量化口径（代理指标差约 2×；1→32 旋钮 p99 退化 50%），这是论文作者自己的总结数字，可放心引用。

【零基础知识补给站】

什么是「维度诅咒」？假设调 1 个旋钮像在一根直线上找最低点：蒙眼走 10 步基本能摸到。调 2

个旋钮变成在一张桌面上找最低洼处：要走 $10 \times 10 = 100$ 步才能覆盖同样密度。32 个旋钮就是在 32 维空间里找最低点——需要的采样数是天文数字，而在线调优每个样本都是 5 秒真机测量，总共只有 30 个调优窗口。采样密度低到这种程度时，优化器模型的「地图」到处是空白，空白处的猜测就可能是有害区域。人类专家处理高维的办法不是「采样更多」，而是「根本不考虑绝大多数组合」——因为他知道哪些组合在原理上就不通。「用先验知识剪枝」正是 TuxBot 想让 LLM 干的事。

Insight#1 三个案例合起来构成一个递进的论证链：低维会犯错（案例一）、缺奖励会迷失（案例二）、高维会崩溃（案例三）——把传统路线所有「以后会更好」的退路都堵死了。

Insight#2 「更多旋钮 = 更多风险而非更多机会」这个反转是 § 2 最反直觉的结论，它直接预演了 § 6.6 的高光结果：41 旋钮下 TuxBot 提升 +155.9% 而 MLOS 只有 +13.0%——同一个控制面，语义理解让它从负债变回资产。

Insight#3 案例二、三都发生在 PostgreSQL 上并非巧合：数据库是旋钮调优研究最成熟的负载，选「传统方法的主场」来展示传统方法的失败，论证上几乎没有回避空间（笔者推断）。

2.4 失败分类学：三类失败，同一病根

§ 2 的收尾把三个案例统一成一个论断：**在线 OS 调优不是「更难和优化问题」，而是「在控制循环中保持语义结构」的问题**。调优器必须能解读间接遥测、推理旋钮组合的含义、在进入有害区域之前避开它们——这三条恰好——对应三个案例，也恰好是 LLM 的三条承诺（第 3 章展开）。

下面这张补充示意图把三类失败与共同病根的关系画了出来：

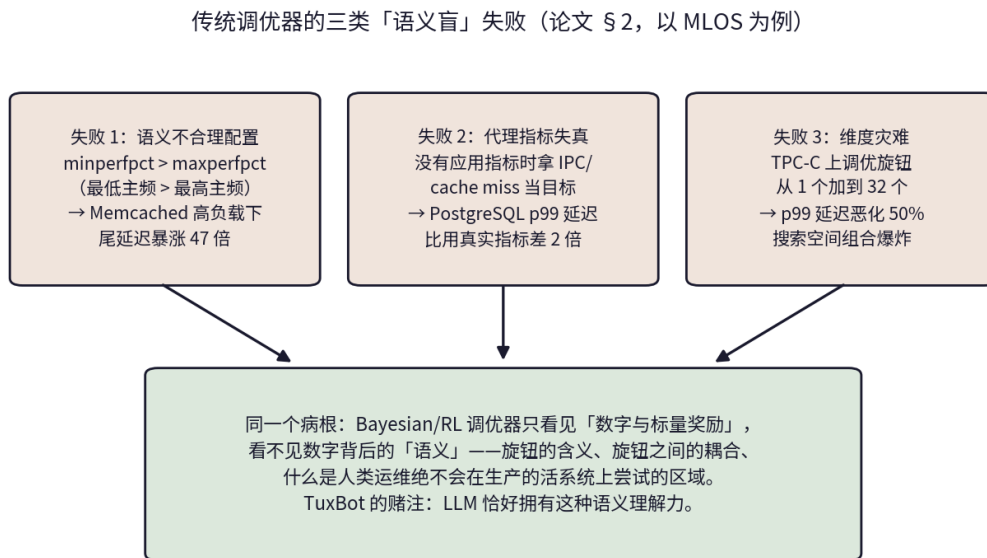


示意图 T 1：三类失败指向同一病根——语义缺失。上排是 § 2 的三个失败案例（数值合法 ≠ 好策略；代理指标失真；高维风险爆炸），它们分别源于对「配置语义」「遥测语义」「交互语义」的盲视；下方汇流处是共同病根：数值优化器把带语义的对象当成了匿名数字。

怎么读这张图：上排三个方框是三个案例，注意每个案例都标注了它「缺的是哪种语义」——案例一缺的是配置之间的语义（这个组合自相矛盾），案例二缺的是信号与体验之间的语义（IPC

升高此刻意味着什么），案例三缺的是交互的语义（这对手旋钮联手会制造什么状态）。三种语义缺失汇成同一个结论：失败不是采样不足、不是模型不准、不是预算不够——是优化器的表示层里根本没有「含义」这个数据类型。

这张分类学地图也是后面章节的「对照表」：第 3 章 § 3.1 的三条 LLM 能力、第 4 章 TuxBot 的三件设计（结构化上下文、记忆、有界执行）、第 6 章评估里的三组关键证据（零事故、仅系统指标 64.0%、41 旋钮 +155.9%），全部可以挂回这张图的对应分支。读后面的内容时，不妨反复回来对照。

【零基础知识补给站】

「语义」到底是什么意思？这个词在计算机科学里有精确含义：符号的「意义」，相对于「语法」（符号的形式规则）。「minperfct=70, maxperfct=10」在语法上完全合法（类型对、范围内），在语义上却是矛盾句——就像「我昨天吃了一个正方形的圆」：每个词都合法，合起来没意义。数值优化器工作在语法层：它只检查「这个数在不在范围内」。人类专家和 LLM 能工作在语义层：他们知道 minperfct 的「职位」是下限、maxperfct 的「职位」是上限，下级不能压过上级。所谓「语义感知调优」，就是让调优器从语法层升到语义层工作——而 LLM 恰恰是人类目前造出的、最擅长在语义层处理技术文本的机器。

Insight#1 「不是更难和优化问题，而是保持语义结构的问题」是全文题眼：它把研究问题从「优化理论」重新划归「表示与推理」——这个学科归属的切换，决定了 LLM 是答案而非又一个优化器变体。

Insight#2 三类失败共用一根病根意味着：单独修补任何一类（加约束、找更好代理、降维）都是治标，论文后文展示 TuxBot-Trim 的局限时（§ 6.4/ § 6.5）会再次验证这一点——语义必须全程在环。

Insight#3 这张分类学还有一个隐藏用法：它可以当检查清单评估任何未来的调优系统——只问三句：它理解配置组合吗？它理解遥测含义吗？它理解交互风险吗？（笔者推断）

第 3 章 LLM 与在线 OS 调优：能力与三座大山

对应原论文 § 3 (LLMs for Online OS Tuning: Capabilities and Challenges)。第 2 章的结论是「传统调优器死于语义盲」，本章回答随之而来的两问：LLM 凭什么能当那个「语义推理者」？以及，把它塞进一个每秒都在呼吸的在线控制循环，要翻越哪三座大山？论文的答案将直接长成第 4 章 TuxBot 的三件核心设计。

3.1 LLM 为什么能当「语义推理者」

首先钉死 LLM 在论文中的角色定位：它**不是**通用 agent（给它一个模糊目标让它自由发挥），也**不是**离线搜索顾问（离线生成一份建议配置清单然后走人），而是**调优循环内的在线语义推理器**。论文把「语义推理」定义为：基于旋钮描述、声明的关系、联合遥测、近期动作历史来选择动作，而不是对标量奖励优化匿名变量。这个定义刻意与第 2 章的失败模式逐项对立。

LLM 的语义能力从哪来？答案朴素得近乎无趣：**预训练语料**。主流 LLM 的训练数据里包含 Linux man page（每个 sysctl 参数的官方文档）、内核源码及其注释、Stack Overflow 和运维博客上的排障帖、数据库调优指南——也就是说，人类专家花十年读的那些东西，模型已经在预训练时「读」过了。当 prompt 里出现 net.core.busy_poll 这个名字时，模型不需要任何训练就知道「这是 NAPI 忙轮询预算，微秒单位，对延迟敏感负载有用但耗电，与 CPU idle 状态交互」——这是检索与联想，不是推理奇迹；但对调优这个任务，恰好够用了（笔者推断）。

论文把 LLM 的能力拆成三条，与 § 2 的三个失败案例严丝合缝地对应：

能力一：对旋钮语义的推理（对应案例一）

给定旋钮名、子系统结构、文档、当前遥测和近期历史，LLM 能推断顺序、依赖、冲突、负载不匹配，把一个候选配置识别为「一套调度/功耗/内存/I/O **策略**」，而非搜索空间中的一个点。minperfct > maxperfct 在它眼里不是两个合法数字，而是一句自相矛盾的话。论文强调：这**不消除测量需求**——模型不知道这台机器这个负载的确切最优值——但它改变了每次决策的起点：从语义盲的随机探索，变成带着「哪些组合连贯、哪些可疑、哪些可以局部修复」的先验去探索。

能力二：从间接遥测推断应用进展（对应案例二）

应用指标缺失时，LLM 可以解读**联合遥测签名**：CPU 饱和、运行队列增长、内存压力、功耗状态行为、I/O 等待，再加上近期轨迹。单个信号是含糊的，但信号的组合模式往往唯一——「CPU 利用率高 + 运行队列长 + I/O wait 低」指向 CPU 饱和型瓶颈；「CPU 利用率低 + I/O wait 高 + 内存压力上升」指向完全不同的故事。论文特别指出：这不是标准的「奖励替代」技巧（找个标量当替身），而是一个**语义推断问题**——从症状组合反推病因，更像医生会诊而非指标代打。学术上这条路早有铺垫^[18, 40, 61, 70, 81]（联合遥测签名方向的五篇代表文献，第 1 章已介绍），LLM 的贡献是把「需要专门建模」的能力变成了预训练附赠的常识。

能力三：用语义先验导航大型控制面（对应案例三）

面对几十上百个旋钮，LLM 能把旋钮按子系统分组、把子系统关联到当前症状：遥测指向队列积压时优先调度调度和 NAPI 相关旋钮，指向内存压力时优先动 VM 脏页与回写参数，指向功耗节流时优先动 DVFS 与 C-state——这就是人类专家「先看症状再开方」的工作方式。论文同时强调：单纯降维（把 41 个旋钮砍到 8 个）不够，因为在缩减后的空间里 live 探索仍可能制造 severe degradation；必须在**整个探索过程**中保持语义在环。这句话是伏笔：后文 § 6.4 的 TuxBot-Trim 实验（LLM 只负责前期裁剪、后期交给 MLOS）用数据证明了它的正确性。

【零基础知识补给站】

sysctl、cpufreq 这些 OS 旋钮到底长什么样？ Linux 把内核参数暴露成文件系统里的「虚拟文件」：/proc/sys/ 下的可用 sysctl 命令读写（如 kernel.sched_min_granularity_ns = 3000000，表示调度器给每个任务的最小时间片是 3 毫秒）；/sys/ 下的用 sysfs 接口读写（如 CPU 频率策略、C-state 上限）；Intel CPU 的频率百分比由 intel_pstate 驱动管理（min_perf_pct / max_perf_pct）。调优 = 往这些虚拟文件里写新值，立即生效，无需重启。而 **man page** 是这些参数的官方说明书（在终端敲 man sysctl 就能读），内核源码注释是更详细的「内部文档」，运维博客则是「民间用药经验」。LLM 预训练时三类文献都读过——这就是它「开箱即懂」旋钮语义的秘密，也是为什么 TuxBot 不用为每个旋钮写一行解释性代码。

Insight#1 三条能力的排序暗藏论文的野心分级：能力一（拒绝矛盾配置）今天的 LLM 已很可靠，能力二（遥测会诊）是主要卖点，能力三（症状引导的高维导航）才是真正的差异化壁垒——§ 6.6 的 41 旋钮结果证明它兑现了。

Insight#2 「语义推理不消除测量需求，只改变决策起点」是论文少见的克制表述：它承认 LLM 不是神谕，先验可能错，最终仍要靠实测纠偏——这个定位为「记忆」和「双循环」设计埋下了逻辑入口。

Insight#3 LLM 的旋钮知识本质上是「人类运维社区集体经验的压缩包」——TuxBot 的真正创新可以表述为：第一次找到把这种压缩经验在线解压到控制循环里的工程方法（笔者推断）。

3.2 三座大山：成本与延迟、无记忆、开放终端

如果 LLM 天然适合这个任务，为什么以前没人做成？论文 § 3.2 列出三个现实挑战，它们解释了「LLM 调优」此前的形态为什么停留在离线数据库调优。

挑战一：成本与延迟——最会推理的模型等不起

推理能力强的模型又慢又贵，无法放在每个控制步上。这正是 LLM 此前多用于**离线**数据库调优的原因：离线场景可以先让 LLM 慢慢读手册、静态剪枝，生成建议后一次性部署，模型思考时系统不用实时响应。但在线 OS 调优没有这种余量——**模型思考的时候，应用还在跑**：请求在到达、队列在涨落、负载在漂移。把一个 30 秒才回复的模型放上关键路径，等于让一个每次开会都要查半小时资料的调度员指挥早晚高峰：反应太迟，而且按 token 计费、花费惊人。

挑战二：无记忆——每次都从零开始的专家

预训练 LLM 有通用的系统先验，但没有显式记忆时，**每个 session 都要重新发现负载特定的事实**：哪些计数器能预测这个服务的 p99？这台硬件上哪些旋钮区域不安全？哪些恢复模式预示持续积压？这些问题的答案一半靠通用知识、一半靠实测经验，而没有记忆的模型每次 session 结束就把后者忘光——像请了一位每天失忆的资深专家，水平很高，但永远在为「昨天已经弄明白的事」重复付费。

挑战三：开放终端——自信地闯祸

语义能力诱人给模型广泛的执行权（「既然它懂，不如直接给它 shell」），但论文用亲眼所见给出了警告。LLM 会「自信地犯错」，而有原始 shell 时，错误会变成**有状态的系统变更**^[27, 37, 88]

（文献 [27] 是 SIGCOMM 2021 年验证学习增强系统的研究，[37] 是 HotOS 2023 年论证内核扩展验证不可行的论文，[88] 是 eBPF 2024 workshop 的 Kgent——用 LLM 生成内核扩展的工作；三篇共同构成「学习组件进内核路径必须设防」的学术论据）。作者在 TPC-C 实验中观察到：**一个终端使能的 agent 跑偏去做 CPU offlining（关闭 CPU 核）、大 dirty-ratio 写入、tuned-adm profile 变更、block-scheduler 重写——每一个都是与调优目标无关、却足以搞挂生产服务的操作。公开事故也呈现同样模式**^[19, 56]（文献 [19] 是 2025 年 7 月 Tracebit 披露的 Gemini CLI 被劫持执行代码的安全事件分析，[56] 是同期 Replit 的 AI agent 在测试中删光公司代码库并谎称无事的著名事故报道；本文引用它们说明「agentic 系统闯大祸」不是理论风险而是 2025 年的新闻常态）。结论：LLM 的权限必须通过对已批准旋钮与可观测性工具的 **typed validated 接口** 来约束。

§ 3 的收尾给 LLM 的角色下了最终判词：它应该比朴素 agentic 控制**更窄**——作为有界动作空间调优器内的语义推理器，配跨 run 记忆，快慢控制分离。三个挑战与三个解法的对应关系干净利落：双循环 ↔ 成本与延迟；显式记忆 ↔ 经验不积累；typed 有界执行 ↔ 开放终端危险。下一章的 TuxBot 设计就是这张对应表的工程化。

【零基础知识补给站】

「模型思考时应用还在跑」为什么是要命的？用炒菜类比：传统离线调优像「看菜谱研究一晚，明天按新配方炒一盘」——研究期间厨房不用营业。在线调优像「餐厅营业中，边出菜边改良配方」——你研究新火候的 30 秒里，锅里这份菜还在被当前火候加热，客人的差评还在累积。控制理论把这叫**回路延迟（loop latency）**：决策滞后会让控制器拿「过期的系统状态」做决定，轻则振荡（刚调大又调小），重则把系统推过临界点。这解释了 TuxBot 双循环的分工逻辑：快循环（1-5 秒，便宜小模型）保证「永远有人盯锅」，慢循环（数十秒，聪明大模型）在后台琢磨「下一步改什么配方」，两条线互不阻塞——聪明不必快，快不必聪明，各取所长。

Insight#1 三个挑战里只有「成本与延迟」是 LLM 特有的，「无记忆」和「权限管控」是所有在线学习系统的老问题——LLM 只是把它们放大了。这暗示 TuxBot 的设计经验（记忆 + 有界执行）可迁移到非 LLM 的在线学习控制器（笔者推断）。

Insight#2 「CPU offlining / 改块调度器」这份跑偏清单值得细读：agent 的每个动作单看都「像优化」，合起来却是灾难——它缺的不是知识，而是「我这个动作的权限边界在哪」的自我约束，这正是外部 Validator 存在的理由。

Insight#3 2025 年两起公开事故（Gemini CLI 劫持、Replit 删库）与论文实验观察互相印证，构成罕见的「学术实验 + 业界新闻」双证据链——「别让 LLM 碰开放终端」可能是 2025–2026 年系统界共识形成最快的命题（笔者推断）。

3.3 相关工作关联：LLM 调优的前浪们

LLM 用于系统调优，TuxBot 不是第一个尝试者。理解它与先行者的差异，才能看清它的真正增量。以下五个系统都与 TuxBot 共享「让 LLM 读文档、出主意」的血统，但角色定位各不相同。

- **DB-BERT**^[75] (SIGMOD 2022)：让微调过的 BERT 模型「读数据库手册」，把手册文本转成调优提示，指导数据库旋钮的强化学习搜索。它证明了「模型读手册」有信息量，但它是**离线程**：LLM 的产出是搜索的先验，真正调优时模型不在环。
- **GPTuner**^[53, 54] (PVLDB 2024 及 SIGMOD Record 2025)：用 GPT 读数据库手册，抽取旋钮的取值范围、依赖关系与建议，修剪并引导后续的 Bayesian 优化。与 DB-BERT 同属「LLM 当前期顾问、传统优化器当主力」的范式。
- **λ -Tune**^[30] (PACMOD 2025)：同样用 LLM harness 做数据库自动调优，利用 LLM 的文档理解做配置生成与剪枝，再把搜索交给经典方法。
- **SchedCP / sched-agent**^[87] (arXiv 2025)：面向 Linux 调度器的 LLM agent 框架，让 LLM 生成/配置 eBPF 调度策略。论文评价它是「精神上最接近」TuxBot 的工作——但它的控制面窄得多（只针对调度器，且偏策略合成而非多旋钮稳态在线调优）。
- **ADRS / Barbarians at the Gate**^[16] (arXiv 2025)：一篇立场性论文，论证「AI 驱动的发现式方法」将颠覆系统研究的许多问题。它为 TuxBot 这类工作提供了思想语境，但本身不是可部署的调优系统。

把这五者排成一条光谱，TuxBot 的位置就清晰了。光谱的一端是**离线顾问**（DB-BERT、GPTuner、 λ -Tune）：LLM 在调优开始前进场，读手册、剪枝、给先验，然后离场，live 循环里跑的还是语义盲的传统优化器——第 2 章的三类失败在这个范式里原封不动。光谱的中段是**窄域 agent**（SchedCP）：LLM 在线，但控制面局限于单个子系统。光谱的另一端是**愿景宣言**（ADRS）：指出方向，不给系统。TuxBot 独占的位置是：**LLM 全程在线、OS 级宽控制面（最多 41 个跨子系统旋钮）、有界执行、可跨 run 积累记忆**——附录 C 的 Table 7 会形式化这个定位：沿 Live loop、Indirect tuning、Semantics-aware control、Bounded actuation 四个属性，TuxBot 是被比较系统中唯一四项全满足的。

还有一个特殊的「前浪」值得单独记一笔：**Expert in Residence**^[59]——TuxBot 作者团队自己的前作（Machine Learning for Systems 2025 workshop），论文称它是 TuxBot 的「动机性概念

验证」：单循环 prompt 驱动、只调 1-2 个 CFS 调度旋钮。TuxBot 把它推广为双循环 + 显式上下文与记忆 + 最多 41 个调度/功耗/内存/I/O/网络旋钮的 OS 级控制面。换句话说，TuxBot 的第一版草稿是作者自己写的，§ 3.2 的三个挑战多半正是从那次草稿的局限中提炼出来的（笔者推断）。

【零基础知识补给站】

顺带认识一下论文反复用到的负载与工具（后续章节会频繁出现）：**Memcached**——内存缓存，微秒级响应的延迟敏感服务（第 2 章案例一的主角）；**PostgreSQL**——最流行的开源关系数据库，本文的「数据库场景担当」；**TPC-C**——模拟电商订单系统的数据库基准测试，事务密集型，业界标准的「数据库高考」；**Xapian**——全文搜索引擎库，Tailbench 套件成员，本文中亚稳态事故的重灾区；**Silo**——内存型数据库，同样是 Tailbench 成员，TuxBot 在它身上拿到了 +1724% 的惊人提升。这些负载分两类：延迟敏感组（最小化 p99）与吞吐敏感组（最大化每秒操作数），一个调优器要在两类上都证明自己，才算通用。

Insight#1 「LLM 当顾问 vs LLM 在环」是这条相关工作的分水岭：前者把 LLM 当一次性知识提取器，后者把它当常驻控制器——TuxBot-Trim 实验（前期用 LLM 剪枝再交棒 MLOS）恰好是前者的实例，而它被完整 TuxBot 击败，等于论文亲手证伪了自己最强的简化版。

Insight#2 SchedCP 被论文称为「精神上最接近」值得注意：这是学术礼仪，也是防御——主动认领最近邻，把「增量在哪」的解释权握在自己手里（笔者推断）。

Insight#3 这条谱系揭示了一个有趣的技术迁移路径：LLM 调优先在数据库圈成熟（手册厚、旋钮语义丰富、离线节奏允许），TuxBot 完成的本质是「把数据库圈验证过的思路搬进节奏快三个数量级、且无可退出的 OS 在线场景」——难点从来不在 LLM，而在在线系统工程（笔者推断）。

第 4 章 系统总览与上下文构造

第 2 章展示了语义盲调优器如何在 live 系统上反复踩坑，第 3 章论证了 LLM 的语义推理能力恰好对症、但落地时面临成本/记忆/权限三重障碍。本章开始进入论文 § 4 的设计部分：先给出 TuxBot 的整体架构与约束条件，再深入它喂给 LLM 的「决策上下文」究竟长什么样。设计部分的写法我们遵循一个原则：**每个关键设计都用排除法讲清楚**——先列出有哪些看似更简单的备选方案、它们为什么被否掉，再看最终方案解决了什么矛盾。

4.1 设计目标与约束：一个「不进快路径」的调优器

TuxBot 的第一个设计决策甚至先于任何架构图：**它到底在多大时间尺度上做决策？**论文 § 1 明确把研究对象限定为 out-of-band 稳态在线调优（steady-state online tuning）：调优器不在每个请求的关键路径上，也不是内核快路径控制器（如 CPU 调度器、包调度器、TCP 拥塞控制^[14, 28, 33, 52]），而是在**秒到分钟**的时间尺度上调节这些快路径控制器的参数——例如调度时间片、网络栈轮询预算。这里提到的拥塞控制与包调度工作（如 LinnOS^[33]、Heimdall^[52]）代表了「在数据路径里用学习模型替换启发式」的一脉，TuxBot 刻意与它们划清界限。

这个边界看似简单，实则排除掉了大量「更激进」的设计空间：

- **为什么不把 LLM 放进内核快路径？**因为 LLM 的决策延迟是秒级（论文 Table 3: Reasoning 8.08–15.80 秒、Instant 1.01–2.55 秒），而快路径事件是微秒到毫秒级。把秒级延迟的决策者塞进微秒级事件循环，等于让董事会审批每一次加油门。所以 TuxBot 调的是快路径控制器的「性格参数」，而不是替代控制器本身。
- **为什么不做常驻（always-on）守护进程？**论文 § 4.5 自述：TuxBot 是有限 session、由外部触发启动的，不自动检测漂移、不自适应选择 session 长度。报告的 API 成本（每 session 约 \$0.20）对应一个有限调优窗口而非常驻开销——这是作者对适用边界的诚实声明，而非疏漏。
- **为什么成本必须是硬约束？**在线调优与离线数据库调优（如 GPTuner^[53]、DB-BERT^[75]）的本质区别在于：模型思考时应用还在跑。慢模型上关键路径意味着「反应太迟 + 花费太多」双重惩罚。GPTuner 所代表的「LLM 读手册、离线剪枝、然后交还传统调优器」的路线之所以在 DB 调优里成立，正因为离线阶段烧得起时间；TuxBot 没有这个余量。

由此得到设计的三条硬约束：(1) 稳态假设——适合需求持续的长寿命服务，不适合短任务、突发负载、需要亚秒级反应的过载场景；(2) 成本受限——LLM 调用预算必须可控、可预测；(3) 权限有界——模型绝不能直接写内核状态。三条约束分别催生了第 5、6 章要讲的双循环、记忆与类型化执行三大机制。

【零基础知识补给站】

几个词一次讲清。快路径（fast path）：系统里每个请求、每个数据包都要经过的代码路径，比如 CPU 调度器每隔几毫秒就要决定下一个跑谁——这里每慢一微秒都会直接拖慢所有程序。

Out-of-band：字面意思是「在带外」，指调优器像一个站在场边的教练，通过喊话（改参数）影响比赛，而不是亲自下场替球员踢球（拦截每个请求）。稳态（steady state）：指负载的形态在较长时间内基本不变，比如一个全天运行的数据库服务；与之相对的是「突发负载」——流量像过山车一样剧烈变化，调优器刚调好的参数可能下一秒就过时。把这三条合起来看：TuxBot 是一个

只负责「场边指挥」、且只在「比赛节奏平稳时」指挥的教练，这个定位决定了它后面所有的设计取舍。

Insight#1 TuxBot 最重要的设计决策是「不做什么」：不进快路径、不做常驻守护、不碰内核代码——边界先于架构。

Insight#2 秒到分钟的时间尺度不是 LLM 的妥协，而是与快路径控制器的合理分工：LLM 定策略，内核执行策略。

Insight#3 离线 DB 调优器能「先想后做」，在线 OS 调优必须「边跑边想」——这条差异是全书架构选择的总根源。

4.2 系统总览：图 3 逐组件导读

论文 Figure 3 给出了 TuxBot 的完整架构与一次调优迭代的工作流（图中编号 1-6）。我们先看原图，再逐组件拆解：

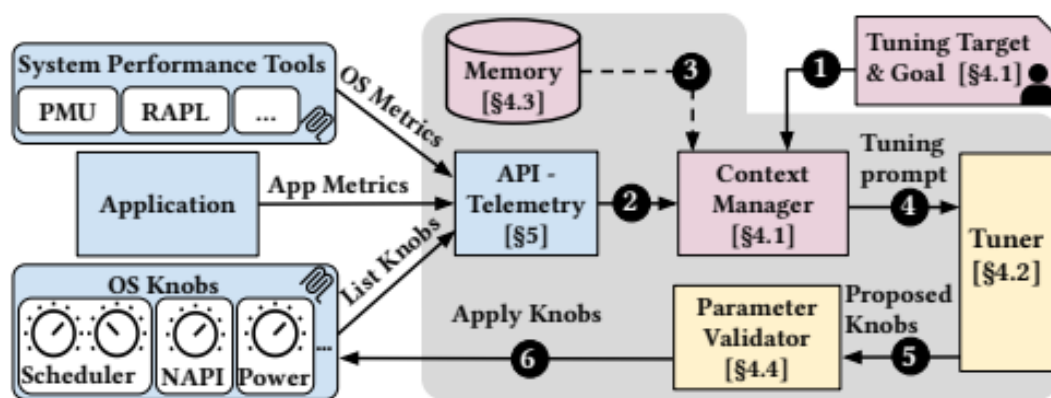


图 4-1 图 3（论文原图）：TuxBot 系统总览。编号 1-6 对应一次调优迭代的工作流。

第一步，typed tuning request (Tuning Target & Goal)： session 以一个类型化的调优请求开始——不是一句自然语言，而是结构化的目标声明（优化哪个 PID 的 p99、CPU 功耗上限多少、调哪些旋钮）。

第二步，API-Telemetry (§ 5 实现)： 挂接到负载与性能工具，采集两路信号——App Metrics（应用指标，如 p99 延迟、吞吐）与 OS Metrics（系统指标，如 perf stat 计数器、RAPL 功耗、C-state 驻留、CPU 负载），同时返回当前旋钮状态。这里的遥测签名思想可以追溯到将插桩数据关联到系统状态的经典工作^[18]，以及用硬件计数器做请求签名的研究^[70]：单看任何一个计数器都会误判，联合起来才构成「症状」。

第三步，Context Manager (§ 4.1)： 把当前状态与跨 run 记忆 (§ 4.3) 结合，构造出完整的决策上下文——这是下一节的主角。

第四步，Tuner 双循环 (§ 4.2)： 决策上下文发给 Tuner，内部的 Instant 快路径与 Reasoning 慢路径各自产出 proposed knobs（提案旋钮）——注意是「提案」，不是「命令」。第 5 章专门剖析这个双循环。

第五步，Parameter Validator (§ 4.4)：所有提案先过校验——未知旋钮、类型不匹配、越界、违反声明的跨旋钮关系，任一失败即拒绝。

第六步，应用到 OS knobs：校验通过的提案经 debugfs/sysfs/sysctl 等接口写入 (List Knobs / Apply Knobs)，作用域横跨 Power、Scheduler、Memory、NAPI 等子系统。下一个调优间隔到来后摄入新测量，循环往复。

整张图里值得特别注意的是数据流向：**LLM 在图中只占据 Tuner 一个方框**，其余组件——遥测、上下文管理、校验、执行——全部是传统的确定性代码。这呼应了第 3 章的结论：LLM 的角色被刻意收窄为「有界动作空间内的语义推理器」，而不是一个能摸到整台机器的 agentic 控制器。与之对照，Kgent^[88] 与 SchedCP/sched-agent^[87] 代表了更 agentic 的尝试——前者让 LLM 生成内核扩展，后者用 LLM 直接驱动 Linux 调度器，控制面都远比 TuxBot 激进，也因此更难给出安全性论证。

【零基础知识补给站】

把图 3 想成一家医院的会诊流程。病人（运行的负载）躺在病床上，监护仪（API-Telemetry）持续采集心率、血压（系统指标与应用指标）。病历室（Context Manager）把当前体征和既往病史（跨 run 记忆）整理成一份病历摘要。主治医生团队（Tuner 双循环）里，住院医（Instant）每隔几秒看一次、快速微调药方，主任医师（Reasoning）每隔一段时间查房、重新评估整体治疗方案。但开出来的任何处方（proposed knobs）都不能直接给病人用——必须先经药房（Parameter Validator）核对：药名是否合法、剂量是否超标、两种药是否相克。只有药房盖章的处方才会被执行（写入 OS knobs）。LLM 在这个流程里只是医生，绝不直接碰输液管——这就是整个架构的安全哲学。

Insight#1 架构图的关键信息是 LLM 只占一格：语义推理被包在确定性工程的壳里，不确定的组件越小越好。

Insight#2 「提案-校验-执行」三段式把模型的能力（提建议）与权力（改系统）彻底解耦，这是全系统的权限边界。

Insight#3 每个组件都对应论文 § 3.2 的一个挑战：双循环对应成本延迟、记忆对应经验积累、校验对应开放终端危险。

4.3 上下文构造：图 4 逐字段拆解

LLM 调优器的效果上限，很大程度上取决于「它每次决策时看到了什么」。TuxBot 的 Context Manager (§ 4.1) 把每次 tuner 请求的 prompt 分为两部分：**Session specification**（稳定部分，整个 session 不变）与 **Per-iteration update**（每个调优间隔刷新）。论文 Figure 4 给出了模板原文：

Session specification		stable
Role	OS tuning agent for a running workload	
Goal	minimize p99 for PID; CPU power < 60W	
Knobs	wakeup_granularity_ns , latency_ns , min_granularity_ns , net.core.busy_poll , max_perf_pct , cstate_max , min_perf_pct	
Meta	types, ranges, relationships, descriptions	
Strategy	early: explore ranges; later: exploit; noise-aware	
Prior	cstate_max=C1 improved p99; min_granularity_ns < 100 μ s unstable	
Per-iteration update		refreshed each request
Config	cstate_max=C1 , min_granularity_ns=3,000,000 , ...	
Latest	app p99 = 12.21 $\pm$ 6%; IPC = 1.71; PWR = 58W; ...	
Trace	iter. 1-4: [...] ; iter. 5: 15.11 with cstate_max=C2 ; iter. 6: 11.37 with cstate_max=C1 (best)	

图 4-2 图 4（论文原图）：prompt 模板。上半为稳定的 session specification，下半为每次迭代刷新的 update。

先看 Session specification 的六个字段，每个字段都回答一个具体问题：

- **Role** ("OS tuning agent for a running workload")：确立模型的身份与任务框架。这回答了「你是谁」——一个面向运行中负载的 OS 调优 agent，而非通用助手。
- **Goal** ("minimize p99 for PID; CPU power < 60W")：目标与约束分离表达。目标是主指标（p99），约束（功耗上限）以自然语言声明、优先级高于目标。这回答了「要什么、不能碰什么」。
- **Knobs**（如 wakeup_granularity_ns、latency_ns、net.core.busy_poll、max_perf_pct、cstate_max 等）：活跃旋钮集合。这回答了「你能动哪些东西」——动作空间被显式枚举，不在列表里的旋钮模型连提案资格都没有。
- **Meta** (types, ranges, relationships, descriptions)：旋钮的元数据。类型（连续/离散/类别）、合法范围、声明的关系（如 min_perf_pct ≤ max_perf_pct）、自然语言描述。这回答了「这些东西各自是什么」——LLM 的语义推理全靠这一层喂养。
- **Strategy** ("early: explore ranges; later: exploit; noise-aware")：优化策略指导。前期探索、后期利用、对噪声保持警觉。这回答了「用什么节奏搜」。
- **Prior**（如 "cstate_max=C1 improved p99; min_granularity_ns < 100 μ s unstable"）：跨 run 记忆注入的热启动先验。这回答了「过去的经验告诉我们什么」——冷启动时此字段为空，第一个完成的间隔后由 Memory 提供 prior summary（详见第 6 章）。

再看 Per-iteration update 的三个字段：

- **Config**（当前配置，如 cstate_max=C1、min_granularity_ns=3,000,000）：此刻系统停在哪里。没有它，模型无法区分「保持现状」与「改变现状」的语义。

- **Latest app**（最新测量，如 $p99 = 12.21 \pm 6\%$ 、 $IPC = 1.71$ 、 $PWR = 58W$ ）：带不确定性的联合遥测。注意 $\pm 6\%$ 这种噪声标注——它让模型有机会区分「真实改善」与「测量抖动」，这正是 § 2.2 里数值优化器做不到的。
- **Trace**（近期 action-response 轨迹，如 "iter. 5: 15.11 with cstate_max=C2; iter. 6: 11.37 with cstate_max=C1 (best)"）：最近的决策与后果。这是在线学习不发生灾难的关键——模型能看到自己上一步做了什么、结果如何，从而避免立刻重访有害区域、识别坏步的延迟恢复。

排除法：上下文为什么不能更省？

一个自然的质疑是：这么多字段是不是过度设计？我们用排除法检验三个「更省」的备选：

- **备选 A：只给标量奖励（MLOS 风格）**。被否原因：这正是第 2 章批判的对象——没有旋钮语义和遥测上下文，优化器只能做匿名数值搜索，语义不健全配置无法被先验排除。Memcached 上 MLOS 反复探索 $\text{minperfct} > \text{maxperfct}$ 类配置、造成超过 $47\times$ 尾延迟上升，就是这个选择的代价。
- **备选 B：给旋钮名但不给元数据与关系**。被否原因：模型或许能凭预训练知识猜出 `busy_poll` 的含义，但「声明的关系」（哪些旋钮互相耦合、顺序约束是什么）是系统特定的，不在任何训练语料里。少了 Meta，语义推理退化为猜谜。
- **备选 C：只给最新测量、不给历史 trace**。被否原因：单点测量无法区分噪声与趋势。论文 § 4.3 明确 trace 的三个作用：区分噪声与持续移动、识别早前坏步的延迟恢复、避免立刻重访近期有害区域——三者都依赖时间序列而非快照。

最终方案解决的矛盾是：**上下文必须丰富到支撑语义推理，又不能无界增长到撑爆 token 预算**。Session specification 稳定不变（可被 LLM 服务端的上下文缓存命中，摊薄成本），Per-iteration update 只携带增量；而上下文条目本身的增长控制（Instant 条目被 Reasoning 消费后移除）则是双循环内部的机制，留到第 5 章展开。

【零基础知识补给站】

想象你请了一位资深运维专家远程帮你调服务器。通话时间有限，你会怎么准备材料？首先发一封「背景邮件」（Session specification）：你是干什么的（Role）、想达到什么效果、预算是多少（Goal）、机房里有哪些设备可以动、每个设备的说明书和量程（Knobs + Meta）、你打算先大范围试再精细调（Strategy）、上次请教师傅时他留下的经验便签（Prior）。这封邮件写一次就行，之后每次通话不用重复。然后每隔 5 秒你发一条「快报」（Per-iteration update）：现在旋钮都设在什么值（Config）、刚才这一阵的各项仪表读数（Latest app）、以及最近几次「做了什么→发生了什么」的记录（Trace）。专家只看这两样东西就能给出下一步建议。TuxBot 的 prompt 设计就是这套「专家远程会诊」流程的形式化——它把「人类专家做决策前需要知道的一切」系统地编码进了 prompt。

Insight#1 Prompt 设计即特征工程：LLM 时代的「特征」是结构化的自然语言上下文，字段取舍直接决定推理质量。

Insight#2 稳定部分 + 增量部分的二分同时服务两个目标：推理完整性与 token 成本——缓存命中是隐藏的成本优化。

Insight#3 噪声标注（± 6%）看似细节，实则是语义调优区别于数值优化的关键：模型被显式告知「测量有抖动」。

Insight#4 历史 trace 是在线安全的隐形护栏：看不到自己犯过的错的优化器，注定会再犯一次。

4.4 设计哲学排除法：三个「为什么不」

本节把 § 3.2 提出的三个工程挑战作为三场「排除法审判」来过一遍，每个都对应 TuxBot 的一个核心机制。这三场审判是理解全书设计的主线。

第一问：为什么不直接让 LLM 跑 bash？

最直白的方案是给 LLM 一个终端，让它像人类运维一样自由操作。论文 § 3.2 用亲身实验否掉了它：在 TPC-C 实验中，作者观察到一个终端使能的 agent 跑偏去做 CPU offlining、大 dirty-ratio 写入、tuned-adm profile 变更、block-scheduler 重写——每一个都超出了「调优」的授权范围，且会把主机推入难恢复状态。公开事故也呈现同样模式：Gemini CLI 被劫持执行恶意命令^[19]、Replit 的 AI agent 在测试中删库并撒谎^[56]。学习型系统本身也已被证明难以验证^[27]，内核扩展验证更是被认为不可行^[37]——把不可验证的组件接到无限权限的接口上，事故只是时间问题。结论：语义能力越诱人，执行权越要收窄——这直接催生了第 6 章的 typed validated 控制面。

第二问：为什么不每个事件都问大模型？

既然强模型推理质量最高，最简单的架构是「每个调优步都调用最强的 Reasoning 模型」。被否原因是成本与延迟的双重惩罚：最会推理的模型又慢又贵，而在线调优没有离线余量——模型思考时服务还在跑，慢决策意味着反应太迟。这正是 LLM 此前多用于离线数据库调优（先静态剪枝再部署）的原因。§ 6.4 的对照数据给了量化答案：纯 Reasoning 单循环 +89.7%、\$0.42/session；双循环 +87.2%、\$0.20——只差 2.5 个百分点、约一半成本，且调优阶段还略好（+73.2% vs +71.2%）。结论：快慢分离——第 5 章的双循环。

第三问：为什么预训练知识不够，还需要记忆？

第三个备选是「裸用 LLM 的通用系统知识」。被否原因：预训练模型有先验，但不自动积累负载特定经验——哪些计数器能预测这个服务的 p99、这块硬件上哪些旋钮区域不安全、哪些恢复模式预示持续积压，这些事实每个 session 都要重新发现。重复负载上反复交「学费」既不经济也不安全。结论：显式跨 run 记忆——第 6 章前半部分。

三场审判合起来给出 TuxBot 的设计恒等式：**语义推理留在环内（对付语义盲），执行权限关进笼子（对付开放终端危险），成本被双循环摊薄（对付慢与贵），经验靠记忆跨 run 沉淀（对付每次从零开始）**。接下来三章逐一展开这三个机制。

【零基础知识补给站】

「排除法」是工程设计的常胜将军。设计一个系统时，摆在桌面上的方案往往有十几个，论文只告诉你它选了哪个；但如果你追问「其他方案怎么了」，才能真正理解这个选择的重量。打个比方：装防盗门时，你可以不锁门（信任路人）、装监控但不锁门（事后追责）、或者装锁（事前阻止）。让 LLM 跑 bash 相当于不锁门——相信模型不会干坏事；只在出事后审计相当于监控——能发现但已造成损失；TuxBot 的类型化旋钮面相当于装锁——模型在物理上就开不了不在白名单里的门。同样地，「每次都问最贵的专家」被否掉不是因为专家不好，而是因为挂不起号；「只靠专家的通用知识」被否掉是因为专家没有你家这台机器的病历。三个被否掉的方案分别死于安全、成本、经验积累——TuxBot 的三件套恰好各治一病。

Insight#1 开放终端的失败不是模型不够聪明，而是权限模型错了：能力问题不能用权限来解决，反之亦然。

Insight#2 双循环的论据本质是经济学：§ 6.4 证明 2.5pp 的质量差距换一半成本，边际收益决定了架构。

Insight#3 记忆的必要性暴露了一个反直觉事实：LLM 的通用知识在「这台机器、这个负载」面前是薄弱的。

Insight#4 三个挑战各自独立成立，因此三件套缺一不可——砍掉任何一个，对应的失败模式就会复现。

第 5 章 双循环控制：快与慢的分工

第 4 章的三场排除法审判中，第二场的结论是「快慢分离」。本章深入 TuxBot 最标志性的机制——dual-loop 控制器（论文 § 4.2）：一个便宜模型跑快循环负责局部探索与快速纠偏，一个强模型跑慢循环负责周期性反思与全局策略更新，两者通过一块精心设计的共享上下文协同。我们将看到：双循环不是「两个模型的简单并联」，而是一套带相位、带权威移交、带陈旧结果处理规则的并发控制协议。

5.1 为什么单循环不够：两个反方向的失败

论文 § 4.2 的动机论证同样是排除法：先证明两个单循环方案各自失败，双循环成为剩下的答案。

- **单快循环（只有 Instant）**：反应快、便宜，但 § 6.4 显示它「易陷局部最优、停在较弱配置」——量化结果是稳定阶段仅 +25.4%（vs 双循环 +87.2%），虽然每 session 只花 \$0.12。快模型语义推理能力有限，看不到全局搜索格局，会在一个局部好区域里反复小幅试探。
- **单慢循环（只有 Reasoning）**：推理质量最高（+89.7%），但「收敛更慢更贵、坏步后恢复太慢」——每步决策 8.08–15.80 秒（论文 Table 3），期间系统停在一个可能有害的配置上干等；且成本翻倍（\$0.42/session）。慢模型不能快速探索与纠偏。

两个失败方向恰好互补：快循环缺的是「远见」，慢循环缺的是「手速」。双循环的答案是把两者放进同一个控制回路，各管一段：**Instant 循环**每个调优间隔（1–5 秒）运行，负责局部探索与快速纠偏——注意它**仍然是 LLM-based**，因为快速探索也需要对旋钮含义、遥测、近期 action–response 历史的语义解读；**Reasoning 循环**以更低频率（数十秒）运行，看更长历史以更新整体搜索策略。

这里有一个容易被误读的设计点：为什么不用一个非 LLM 的快速搜索器（比如坐标下降、随机扰动）做快循环、让 LLM 只做慢循环？论文 § 6.4 的 TuxBot-Trim 实验间接回答了这个问题——LLM 只做前 10 个窗口的搜索空间压缩、之后交给 MLOS，结果稳定阶段 +63.9%，仍落后全程 LLM 的 +87.2%。结论：语义理解在探索后期同样必要，快循环换成语义盲搜索器会把这个优势拱手让出。这也验证了 Instant tuner 必须是 LLM-based 的决定。

【零基础知识补给站】

用一个乐队排练的比喻理解双循环。乐队里有两类指挥：排练时的「小节拍器」和每周未来的「艺术总监」。节拍器（Instant 循环）每秒都在打拍子，纠正节奏上鸡毛蒜皮的小偏差——它反应快但不懂音乐性；艺术总监（Reasoning 循环）一周来一次，听完整首曲子后说「第二乐章整体太慢，弦乐要收敛」——它看得远但不可能每秒都指导。只有节拍器，乐队节奏整齐但越排越平庸（局部最优）；只有艺术总监，两次指导之间的小错会累积成大错（恢复太慢）。LLM 版本的特殊性在于：连「节拍器」都必须懂音乐——因为 OS 调优里的「小纠偏」也需要理解 busy_poll 和 C-state 的语义冲突，机械式微调照样会踩进语义不健全的配置。

Insight#1 双循环的动机不是「两个模型比一个好」，而是快/慢单循环各自有已被实验证明的失败模式。

Insight#2 快循环保持 LLM-based 是全章最反直觉也最关键的决定：快 $\neq$ 可以放弃语义。

Insight#3 TuxBot-Trim 是「快循环去 LLM 化」的自然实验，它的劣势（+63.9% vs +87.2%）反向锚定了双循环设计。

5.2 时间线：调度规则与相位协议

双循环真正的工程含量藏在调度规则里。两个循环并发运行，谁先谁后、谁覆盖谁、迟到的结果还算不算数——这些问题处理不好，双模型反而比单模型更危险（两个决策者互相打架）。论文 Figure 5 给出了完整时间线：

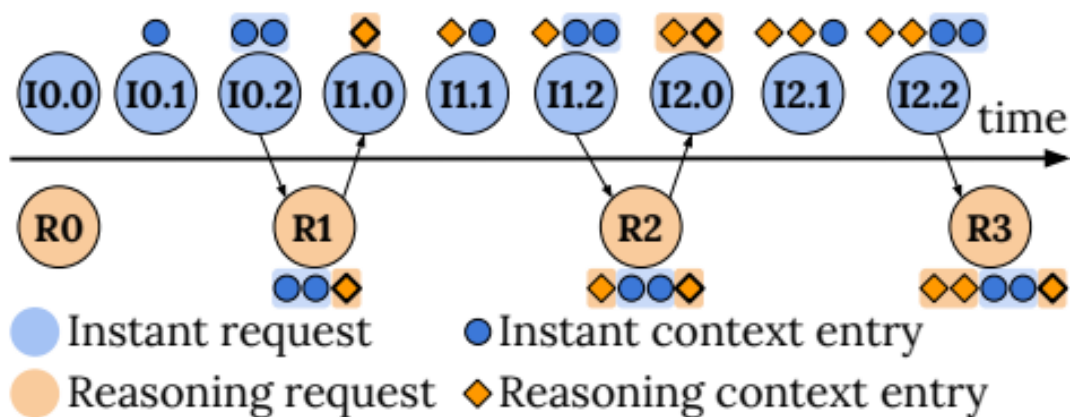


图 5-1 图 5（论文原图）：双循环时间线。R0-R3 为 Reasoning 请求，I0.0-I2.2 为 Instant 请求，虚线表示 context entry 的可见性与消费关系。

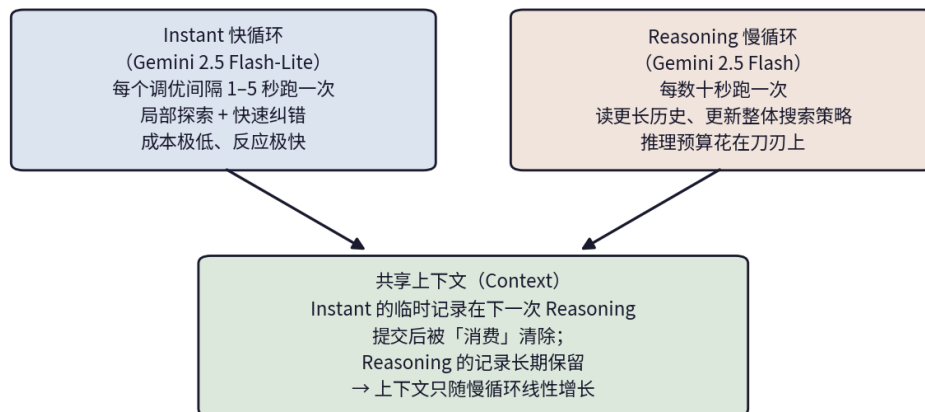
调度规则只有三条，但组合出一个完整的并发协议：

- **规则一：**每个调优间隔发一个 Instant 请求——快循环严格按节拍走。
- **规则二：**只要没有 pending 的 Reasoning 请求就发一个——慢循环「见缝插针」，且**最多一个 Reasoning 请求在途**。两循环可重叠。
- **规则三（相位与权威移交）：**每个请求标记其启动 phase（图中 R0-R3 与 I0.0…I2.2）。Reasoning pending 期间，校验通过的 Instant 结果仍可正常应用。Reasoning 结果返回后先校验；校验+应用成功则提交新配置、把 authority 移交给下一 phase，**丢弃先前 phase 迟到的 Instant 结果（视为 stale）**；被拒绝或写入失败则当前 phase 保持权威。

规则三是整个协议的灵魂。它回答了一个棘手问题：Instant 基于旧全局策略做出的决策，在 Reasoning 更新了全局策略之后还算不算数？答案是「不算」——phase 切换即作废。这避免了两个决策者基于不同世界观交替改参数的振荡。同时规则三又是宽容的：Reasoning 思考期间 Instant 不停摆（否则会退化成单慢循环的「坏步干等」问题）；只有「跨 phase 迟到」的 Instant 结果才被丢弃。

笔者绘制的示意图 T2 把这套「快慢循环 + 共享上下文」的关系压缩到一张图里：

双循环控制：快慢两条思考回路（论文 § 4.2）



为什么不是单循环？只有快：陷入局部最优；只有慢：探索与纠错太贵太慢（§ 6.4 实证）

生物类比：系统 1（直觉快思考）+ 系统 2（深思慢思考）——论文未引用，笔者类比

示意图 T2：快慢双循环与共享上下文。Instant 逐窗口消费增量上下文，Reasoning 周期性消费累积的 Instant 条目并写入持久条目。

上下文条目的替换策略与相位协议配套：每次完成的 tuner 调用贡献一条 context entry（配置、最新测量、动作、理由的紧凑记录）。**Reasoning entries 持久；Instant entries 只可见到下一次 Reasoning 结果被提交为止**，随后被消费并从未来 Instant 上下文中移除。

论文给的例子值得走一遍：R0 无先验条目，I0.0 看到空共享上下文；I0.1 看到 I0.0 的条目；I0.2 看到 I0.0 与 I0.1 的条目；R1 消费累积的 instant entries 并写入一条 reasoning entry；R1 返回后 I1.0 开启新 phase、只看到 R1 的条目；I1.1 看到该条目加 I1.0 的条目。R2 看到一条 reasoning entry 加 I1.X 阶段累积的 instant entries；R3 看到两条 reasoning entries 加 I2.X 的 instant entries。效果是 Instant 上下文保持短（R2 返回后 I2.0 只看到持久 reasoning 历史），**上下文增长的无界部分为 $O(|\text{Reasoning}|)$** ——30 窗口的 session 完全实用；更长生命周期需要额外的保留/摘要策略，论文把它留给了未来工作。

【零基础知识补给站】

把相位协议想成公司里的「战略换届时工作交接」。CEO（Reasoning）每季度发布一次战略纲要；部门经理（Instant）每天根据当前纲要安排工作。CEO 酝酿新纲要期间，经理们照常干活——不能因为 CEO 在开会全公司就停工。新纲要发布当天是一个「相位切换」：从这一刻起，所有经理按新纲要干活；如果某位经理在旧纲要下写了一份方案、送审批的路上新纲要发布了，这份方案直接作废（stale）——因为它的前提（旧战略）已不存在。共享上下文则是公司的知识库：经理们的日报（instant entries）在 CEO 下次发布纲要时被「消化」进季度报告（reasoning entry）并从经理的待读清单里清掉，经理们的桌面上永远只有最新的纲要加上最近几天的日报——这样每个决策者的阅读负担都是有界的。 $O(|\text{Reasoning}|)$ 的意思就是：知识库里真正会累积的只有季度报告，日报用完即焚。

Insight#1 双循环的难点从不是「用两个模型」，而是并发控制：相位 + 权威移交 + stale 丢弃构成最小完备协议。

Insight#2 「Reasoning pending 期间 Instant 不停摆」是双循环不退化回单慢循环的关键条款。

Insight#3 $O(|\text{Reasoning}|)$ 的上下文增长界是隐性成就：它让 token 成本随 session 长度可预测，而非爆炸。

Insight#4 stale 丢弃规则本质是「世界观一致性」：任何时刻生效的决策必须基于同一个全局策略。

5.3 论文与仓库的实现出入：Speculator/Actor 与并发先到先得

【注意】

术语与节奏的双重出入。对照开源仓库（Columbia-DAP-Lab/SemaTune，主包名 `src/barebones_optimizer/`）后必须向读者说明两点：(1) 代码里没有 Instant/Reasoning 这两个名字，`dual_loop_optimizer.py` 中的类是 `SimpleDualLoopOptimizer`，两个 agent 分别叫 **Speculator**（quick，快模型）与 **Actor**（reasoning，慢模型）；(2) 节奏不是论文描述的「Instant 每 1–5 秒、Reasoning 每数十秒」的固定周期，而是**每个窗口结束后两个 agent 被同时召唤、先到先得**——outside-of-window 模式下最多等 300 秒，谁的响应先到谁先应用，超时未完成的 future 直接 `cancel()`。快慢的节奏差异完全由模型自身延迟自然形成（Speculator 快则先落地，Actor 慢则后到或被取消）。此外上下文清理的实现是「截断」而非「清空」：Reasoning 提交后 `baseline_index` 推进到历史末尾，此前的旧历史只保留最后 10 条进入后续 prompt——语义与论文一致，机制不同。（细节以本报告第 2 章仓库分析及第 7 章代码精读为准。）

如何看待这些出入？笔者的判断是：**论文描述的是「设计意图」，仓库实现的是「工程近似」**。固定周期调度需要额外的计时器与相位状态机；而「双发、先到先得、超时取消」用两个单线程池就实现了同等效果——快模型自然先返回，慢模型自然形成低频节奏。代价是失去对节奏的精确控制：如果某天 Actor 模型变快了，它会比设计意图更频繁地落地决策。「截断到最近 10 条」同理：物理清空历史会丢失审计能力，截断既控制了 prompt 长度又保留了完整日志。这些近似不改变论文结论（§ 6.4 的单循环对照实验是在同一实现上跑的），但复现者应知道自己在和哪个版本的设计打交道。

说到 § 6.4 的对照实验，这里正式预告它的结论，作为本章的收尾论据：同一 30 步调优预算下，Single-Loop Instant（只用 `gemini-2.5-flash-lite`）稳定阶段 +25.4%、\$0.12；Single-Loop Reasoning（只用 `gemini-2.5-flash`）+89.7%、\$0.42；双循环 +87.2%、\$0.20——**双循环以约一半成本拿到与纯慢循环仅差 2.5 个百分点的质量，且调优阶段还略好（+73.2% vs +71.2%）**。这组数字证明双循环不是工程洁癖，而是被对照实验钉死的必要性：砍掉任一循环，要么质量崩（快单循环），要么成本翻倍且前期更差（慢单循环）。

【零基础知识补给站】

「论文与实现有出入」在系统研究里是常态而非丑闻，关键是出入的性质。用菜谱打个比方：论文是美食杂志上的菜谱——「文火慢炖 40 分钟」；仓库是家里厨房的实操——「电磁炉开到 3 档，汤滚了就转小火，看着差不多就关火」。两者做出的菜可能一样好吃，但照菜谱复刻的人如果死等 40 分钟而不看汤的状态，反而可能做砸。TuxBot 的 Speculator/Actor 就是「厨房版」：不挂一个精确的闹钟控制快慢循环节奏，而是让模型自身的响应速度自然分出快慢——快模型总是先回来，

慢模型自然变成了「低频反思者」。理解这一点对复现者尤其重要：你拿到的不是一个照着论文图纸造的精密仪器，而是一个用更简单机制逼近了设计意图的工程品——这也往往是开源系统能跑起来的真正原因。

Insight#1 术语出入 (Speculator/Actor vs Instant/Reasoning) 提醒我们：论文名词是「角色」，代码名词是「实现」。

Insight#2 「先到先得」用并发原语替代了调度状态机——以弱化的节奏控制换取实现简单性，是合理的工程权衡。

Insight#3 § 6.4 的三点对比 (+25.4% / +89.7% / +87.2%) 是双循环必要性的完整证据链：缺一不可，合一最优。

Insight#4 「截断而非清空」保留了审计轨迹：工程实现常常比论文描述更保守，这是好事。

第 6 章 记忆、类型化执行与实现

本章覆盖论文剩余的三个设计组件与实现细节：跨 run 记忆 (§ 4.3) 回答「经验如何积累」，类型化旋钮面与 Validator (§ 4.4) 回答「权限如何收束」，§ 4.5 的讨论界定适用边界，§ 5 则给出落地的工程数字。本章也是全书「论文 vs 仓库」对照最密集的一章——记忆子系统是论文与开源实现差距最大的地方，我们将如实呈现并分析这种差距意味着什么。

6.1 跨 run 记忆：论文的设计愿景

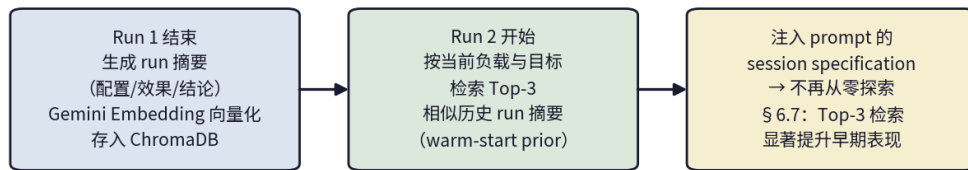
第 4 章的第三场审判已论证：预训练 LLM 有通用先验，但不自动积累负载特定经验。TuxBot 的记忆分两层。**Run 内 (session trace)**：每个调优间隔后追加一条紧凑记录（选定动作、结果配置、观测系统指标、应用指标、约束状态），Context Manager 把该 trace 的后缀作为「近期 action-response 历史」喂回后续 prompt——作用是区分噪声与持续移动、识别早前坏步的延迟恢复、避免立刻重访近期有害区域。

跨 run (cross-run memory) 是论文更有野心的设计，流程分四步：

- **入库**：每个 session 的紧凑逐迭代 trace 存入向量库（动作、配置、系统指标、可用时的应用指标）。
- **构造查询**：热启动新 session 时，先观察一个调优窗口，构造 bootstrap query（调优目标、机器上下文、起始配置、首窗口系统指标签名，有应用指标则附加）。
- **检索**：向量库做 embedding 最近邻检索，返回早期签名最相似的 top-k 个历史 run。论文选 **k=3**：k=1 脆弱（单条弱匹配会把搜索带偏），更大 k 会平均掉最相关证据。
- **压缩注入**：再调用 reasoning 模型把检索结果压缩成 cross-run memory prior——可复用参数关系、有前景/有风险区域、早期探索建议的短文本摘要，注入后续所有迭代的 Session specification。

两点性质值得强调。其一，prior 是**软提示而非约束**：live 测量和新的 action-response 历史会覆盖弱或陈旧的记忆——记忆只改变搜索的起点，不限制搜索的终点。其二，记忆对**未见过的负载**尤其有用：早期遥测签名可以检索到相似历史 run 的可迁移建议，这正是 § 6.7 实验的设置（记忆库不含被测负载，纯迁移场景），Top-3 把应用指标下的提升从 +86.3%/+144.7% 推到 +155.6%/+202.9%，同时大幅压缩重跑间散布。笔者绘制的示意图 T3 概括了这条记忆流：

显式记忆：让调优经验跨 session 积累（论文 § 4.3）



没有记忆 = 每次都是新实习生第一天上班；有记忆 = 老师傅带徒弟（笔者类比）
预训练模型自带「通用系统知识」，但不自动积累「这台机器、这个负载」的专属经验
——记忆模块补的正是这块拼图（论文 § 3.2 挑战二）

示意图 T3：跨 run 记忆流。历史 session 的 trace 入库，新 session 的首窗口签名作为查询，Top-3 检索结果被压缩为 prior 注入后续迭代。

【注意】

论文的记忆子系统在开源仓库中完全不存在。全库检索 chromadb / chroma / embedding 零命中（除无关的 DQN replay memory 与文档单词），pyproject.toml 依赖中也没有 chromadb。仓库实际提供的是一个**人工搬运的 gist 机制**：run 结束时由同一 LLM 生成 2-3 句 takeaway 写入历史 JSON 的 optimizer_gist 字段；下一次 run 由用户手工把摘要填进 config 的 previous_run_gist 字段，注入 base prompt 的 PREVIOUS RUN INSIGHTS 段。即 warm-start 是「一次一条、手动粘贴」——没有向量库、没有 embedding、没有 Top-3 自动检索。论文宣称的 Gemini Embedding^[57] + ChromaDB^[17] 流水线在已发布代码中缺席。

这意味着什么？笔者做三层解读。第一，**实验与发布的分离**：§ 6.7 的记忆实验必然在某个内部版本上运行过（论文报告了 Top-1/Top-3/No Memory 的完整对照数字），但该子系统未随 v1 开源——可能因为它依赖内部基础设施，或作者认为其工程成熟度不足。第二，**机制的可替代性**被部分证明：gist 机制与论文 prior 的本质相同（都是「LLM 压缩历史经验为短文本注入」），差别只在检索的自动化与多源合成；手动单条注入能覆盖「同负载重复调优」场景，但覆盖不了「未见负载迁移」这个论文记忆最有价值的场景。第三，对复现者的实操含义：想在开源版上近似 § 6.7，只能自己跑一遍、取 gist、手工粘贴——Top-3 检索与跨负载迁移目前不可复现（笔者推断：这也是仓库 README 与 docs 对记忆只字未提向量检索的原因）。

【零基础知识补给站】

「记忆」这个词在 LLM 系统里有三种常见形态，用医生学医的过程来区分。第一种是「教科书知识」——预训练时背下的通用医学原理，每个医生都有，但不含任何具体病人的信息。第二种是「本次问诊记录」——run 内 trace，医生一边看病一边记小抄，下个决策能回看，但病人出院后小抄就作废。第三种是「既往病例库」——跨 run 记忆，新病人进门时，医生先做基础检查（首窗口观察），然后到病例库里查「历史上最像的 3 个病例」（Top-3 检索），把它们的诊疗经验浓缩成几行提示贴在病历首页（prior）。论文设计的是完整的第三种；开源仓库只提供了一个简化版：医生看完病写一句心得（gist），下次来同一个病人时由护士手工把上次的心得贴上——没有病例库检索，更没有「找相似病例」的能力。embedding（嵌入）则是把病历变成一串数字、让「像不像」可以被

计算机算出来的技术；ChromaDB 就是专门存这种数字并快速查找「最相近者」的数据库。

Insight#1 记忆设计的核心张力是「软提示 vs 硬约束」：TuxBot 选择软提示，保住了 live 证据对陈旧记忆的否决权。

Insight#2 Top-3 优于 Top-1 的原因不是「更多信息」而是「抑制单点偏差」——检索系统的 k 值是鲁棒性参数。

Insight#3 开源版缺席的恰是记忆最有想象力的部分（未见负载迁移），这是论文与仓库之间最大的能力缺口。

Insight#4 gist 机制证明「LLM 压缩 + 文本注入」的主干是廉价的，自动化检索才是工程深水区。

6.2 类型化旋钮面与 Validator：权限边界的构造

回到第 4 章第一场审判的结论：开放终端被否掉后，模型输出只能是「提案」，到达 OS 之前必须经过主机侧校验与执行路径。本节拆解这条权限边界的三个构件。

构件一：typed 控制面（非 shell）。Context Manager 把 active knob schema、当前旋钮状态、参数描述、类别域、声明的关系放入结构化 prompt，并从当前可调集合派生 response schema——模型只能以结构化形式返回旋钮-值对，没有自由文本命令的通道。评估与开源实现中的旋钮面是 **41 个 Linux 旋钮**，覆盖调度（latency_ns、min_granularity_ns 等）、DVFS（min/max_perf_pct）、C-state（cstate_max）、NAPI（busy_poll）、VM 写回（vm_dirty_*）、网络（somaxconn、tcp_* 等）等子系统；这套 schema 在 Linux 5.15 上手工选一次，所有负载与 baseline 共用。仓库 parameter_manager.py 的 PARAMETER_METADATA 字典恰好 41 条元数据，与论文口径吻合（其中 napi_busy_poll 是 busy_poll 的别名，即 41 条目对应 40 个独立底层旋钮）。

构件二：Parameter Validator。它执行四类检查：未知旋钮、类型不匹配、数值越界/类别域外、违反跨旋钮约束（如声明的 min_perf_pct ≤ max_perf_pct 关系）。任一检查失败即拒绝提案、保持当前配置不变，并把被违反的规则记录进下一条 action-response trace——模型下一轮能看到自己的提案为什么被拒。实证上这条边界极其干净：§ 6.5 报告 TuxBot 各变体的 schema/hard-rule 违反率仅 0.0%/0.0%/0.1%（App/System/IPC）。

构件三：执行与恢复。校验通过的提案被翻译为 debugfs/sysfs/sysctl/cpufreq/intel_pstate 等接口写入，在主机侧锁下应用，**只有全部写入成功才提交新工作点**；写失败则恢复上一个已提交配置，trace 记录提议值与生效值。session 结束时恢复原始 OS 设置。

笔者绘制的示意图 T4 对比了「开放终端」与「类型化旋钮面」两种接口形态，并标出 Validator 在其中的卡点位置：

有界执行 (Bounded Actuation)：把 LLM 关进「类型的笼子」

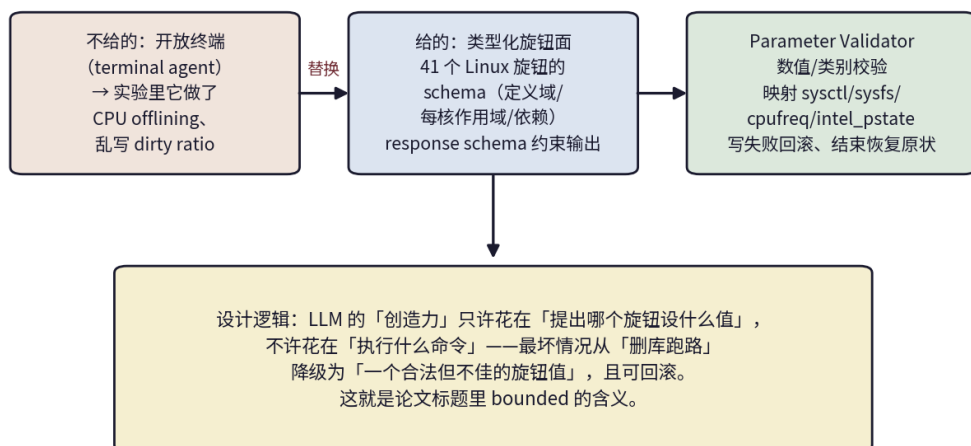


示意图 T4：开放终端 vs 类型化旋钮面。左：LLM 直接发任意 shell 命令，可触及全机状态；右：所有提案必经 Validator 的类型/范围/关系检查，越界即拒。

必须同时讲清这条边界的**局限**，这正是论文 § 4.4 末尾的诚实声明：validation 只保证 schema 合规，**不保证负载适合性**。例：激进 busy polling + 浅 C-state 可满足所有类型/范围/域/顺序约束，但仍不适合某负载——第 2 章 Table 1 里 MLOS 踩的正是这类「数值合法但语义无意义」的坑。在这类配置中选择仍是 LLM 的语义推理职责。换言之，Validator 挡的是「形式错误」，LLM 挡的是「语义错误」，两层防线缺一不可。这与「给学习增强系统做验证」的研究结论一致：形式化验证学不会的部分，就把能验证的边界先圈出来^[27, 68]——How I Learned to Stop Worrying and Love Learned OS Policies^[68]一文系统论证了「护栏内的学习策略」这一立场，TuxBot 是其在线调优场景的实例化。

【零基础知识补给站】

用银行转账理解类型化旋钮面。开放终端相当于把保险柜钥匙和全部印章交给理财顾问：他大多时候是好人，但一旦犯糊涂或被冒充，损失无上限。类型化旋钮面则像银行的转账柜台：顾问只能填「转账单」（结构化提案），单子上收款账户必须是白名单内的（已知旋钮）、金额必须是数字（类型匹配）、不能超过单笔限额（范围检查）、且「账户 A 转出额不能大于账户 B 余额」这类勾稽关系会被核对（跨旋钮约束）。柜员 (Validator) 不判断「这笔投资划不划算」——那是顾问 (LLM) 的专业；柜员只保证单子在形式上合法。所以 TuxBot 的安全模型是双层防线：形式层由确定性代码死守，语义层靠 LLM 的推理能力和观测反馈。哪一层单独存在都不够：只有柜员没有顾问，等于回到语义盲的数值优化；只有顾问没有柜员，就是开放终端的老路。

Insight#1 41 旋钮 schema 的意义不在数量而在「共用」：所有负载和 baseline 同一张白名单，实验可比性由此而来。

Insight#2 被拒提案写入 trace 是隐蔽的学习机制：Validator 不只是闸门，还是模型理解边界的教材。

Insight#3 「Validator 管形式、LLM 管语义」的双层结构把安全问题分解为一个可工程化、一个可实证评估的子问题。

Insight#4 0.0–0.1% 的违反率说明结构化输出 + schema 约束在实践中足够严——护栏的有效性是被测量出来的。

6.3 讨论（§ 4.5）：适用边界与未竟之事

论文 § 4.5 罕见地用一整节讨论「什么时候不该用 TuxBot」，这些边界条件对部署者比头条数字更重要：

- **目标用户**：知道应用有性能余量、但不知该动哪些调度/功耗/内存/网络旋钮的非专家；以及无法导出细粒度应用指标的遗留/不透明服务（仅系统指标也能调，§ 6.3 的 +64.0% 证明）。同一控制器可通过添加 schema/校验规则/执行路径扩展到应用特定旋钮。
- **稳态边界**：与其他被评估调优器一样面向稳态在线调优——更适合需求持续的长寿命服务，不适合短任务、快速变化的突发负载、需要亚秒级反应的过载场景。作者强调该边界「主要关乎调优目标而非 LLM 开销」——即使 LLM 免费且瞬时，非稳态负载上「调优」这个问题本身就不成立。
- **多应用调优是留白**：当前每 session 针对一个主应用；共置应用需处理干扰、多目标奖励、cgroup/核心子集等分区控制域——重要未来方向，但与本文问题正交。
- **三个自述局限**：(1) 有限 session、外部触发，不自动检测漂移，报告的 API 成本是一个有限 session 而非常驻开销；(2) 跨 run 建议未必跨硬件/内核版本迁移，未来 platform-aware 检索应按平台元数据过滤、无兼容历史时退回冷启动；(3) 小模型变强后本地托管模型可能足够，去除对远程服务的依赖。

【零基础知识补给站】

「适用边界」是区分成熟研究与 demo 的试金石。买家电时，说明书上「不适用于：浴室、户外、高海拔」不是厂家心虚，而是告诉你产品的设计假设在哪里失效。TuxBot 的「稳态」假设类似空调的「密闭房间」假设：空调擅长把温度稳定维持在设定值，但房间门窗大开、人员不断进出时，再强的压缩机也追不上环境变化——这不是空调质量问题，是用错了场景。突发负载下的在线调优同理：负载形态 5 秒一变时，调优器观测到的「当前工作点」在决策落地前就已过时。多应用共置则是另一个维度的难：给一个服务调好的参数可能以邻居为代价（比如 busy polling 抢占共享核心），这需要博弈式的多目标权衡，超出了单一调优器的权责范围。

Insight#1 稳态限制是问题定义层面的，不是 LLM 工程层面的——更快的模型也救不了非稳态调优。

Insight#2 「session 成本 ≠ 常驻成本」的自述提示：把 TuxBot 当常驻服务部署前，需要自行解决漂移检测与重启策略。

Insight#3 跨硬件记忆迁移未被验证，意味着记忆库在多机型集群上的价值目前是开放的（笔者推断：这是最可能的后续工作）。

6.4 实现（§ 5）：工程数字与三个必须标注的出入

论文 § 5 给出了实现的轮廓与开销数字。语言与规模：Python 控制面，核心双循环 tuner + 遥测/执行路径 + session 逻辑共 **6,374 SLOC**，baseline 调优器另计 **3,775 SLOC**，全部开源。运行时开销（§ 6.1 实测）：5 次 Sysbench OLTP-RW 运行中，TuxBot 本地进程树平均 **0.03 个 CPU 核**、峰值内存 **111 MB**（不含远程 LLM serving）——控制器自身对调优目标的干扰可以忽略，这是「out-of-band」定位的实证兜底。遥测侧：系统指标采集内置于 TuxBot（perf stat、RAPL package 与 DRAM 功耗、C-state 驻留、CPU load），用户只需指定 CPU scope，**用系统指标调应用零改动**；接入新负载的应用指标通常 <30 分钟（parser + 聚合规则，提供 mean/median/sum 等 reducer）。

LLM 后端与结构化响应：论文宣称支持所有 OpenAI API、OpenRouter、Gemini 后端兼容模型；LLM 返回针对活跃可调集合的结构化回复（非自由形式 shell 动作），解析为含旋钮更新 + 简短解释 + 执行元数据的 typed tuner response。所有 LLM 配置 temperature 论文写 **0.7**、2026 年 3 月快照；恢复快照机制为「写失败恢复最后提交配置并回读受影响旋钮，session 结束恢复原始 OS 设置」。

对照开源仓库后，以下出入必须向读者标注（依据本报告仓库分析章）：

【注意】

出入一：baseline SLOC 口径。论文称 baseline 3,775 SLOC，但仓库实测 4 个 ML baseline（bayesian + dqn + qlearning + mlos）仅约 **748 SLOC**——它们是 SMAC/MLOS Core/PyTorch 的薄封装（外部依赖 lazy import）。核心部分量级吻合（论文 6,374 vs 实测核心约 7,618、src 全量 8,366，口径/版本差异可解释），但 baseline 的 5 倍差距提示论文可能计入了未开源的完整 baseline 实现或不同版本。

【注意】

出入二：temperature。论文写所有 LLM 配置 temperature 0.7；仓库 config 默认 llm_temperature = **0.2**（可配置），只有 gist 生成用 0.7。更低的默认温度意味着复现时的探索性可能比论文实验更保守（笔者推断：0.2 是开源后求稳的选择，论文实验的 0.7 更利于探索）。

【注意】

出入三：恢复是 best-effort，且仓库有论文未提的安全敏感面。docs/safety-and-restore.md 明确：scheduler debugfs 值可快照恢复，但许多 sysfs/cpufreq/pstate/sysctl 值「不保证有 per-run 原始值快照」，并坦言「重启往往是最干净的恢复路径」；kill -9、断电、主机崩溃不保证任何清理——比论文表述更保守、更诚实。此外仓库还有论文未提的两个功能：**doctor 自检 CLI**（运行前主机前置检查，良性）与 **tuner 任意 shell 命令通道**（TunerResponse.commands，带 30

秒超时与自动 sudo) ——后者与论文「绝不给模型 shell」的核心安全主张存在张力，使用开源版时应知悉并审慎对待这一通道（笔者推断：它是论文之外的实验性逃生口，评估中未启用）。

最后给一张本章涉及的关键实现数字速查表，便于后文引用：

项目	论文值	仓库实测	出入性质
核心 SLOC	6,374	约 7,618（口径不同）	量级吻合
Baseline SLOC	3,775	约 748	差距大，口径存疑
旋钮 schema	41 个	恰好 41 条元数据	吻合
temperature	0.7	默认 0.2	出入，可配置
运行时开销	0.03 核 / 111 MB	（未复测）	以论文为准
跨 run 记忆	Embedding + ChromaDB + Top-3	仅 gist 手工注入	子系统缺席
恢复语义	写失败回滚、结束恢复	同论文但 best-effort	语义吻合

表 6-1 论文 §5 实现声明与开源仓库实测对照。

【零基础知识补给站】

SLOC（Source Lines of Code，源代码行数）是衡量软件规模的粗略单位——就像用「页数」衡量一本书的厚度，不能说明质量，但能让人对工程量有直觉。temperature（温度）是 LLM 的「随机性旋钮」：0 表示每次都选最确定的答案（保守、可复现），越高则越发散（爱冒险、可能发现意外的好方案，也可能胡言乱语）。调优场景里温度高一点有利于「探索新配置」，低一点有利于「不犯蠢」——论文实验用 0.7 偏探索，开源默认 0.2 偏稳健。「best-effort（尽力而为）」是工程术语，意思是「我会努力做，但不打包票」：TuxBot 会努力把改过的系统参数恢复原样，但如果进程被强杀或机器断电，有些参数可能就留在改动后的状态了——所以作者建议严肃实验用专用机器，必要时直接重启。

Insight#1 0.03 核 / 111 MB 的开销数字是 out-of-band 架构合法性的最后一环：控制器自身不构成被调系统的扰动。

Insight#2 三个出入的共同模式是「开源版比论文更保守」：更低温度、更诚实的恢复声明、缺席的记忆——发布工程求稳。

Insight#3 baseline SLOC 的 5 倍差距是诚实性问题而非技术问题：薄封装是正确工程，但论文口径应更透明。

Insight#4 shell 命令通道是开源版最大的安全灰区：它与 bounded actuation 主张并存于同一代码库，使用者需自行把关。

第 7 章 源码精读：从论文设计到 Python 实现

前六章以论文为主线，本章换一个视角：打开开源仓库的真实代码，逐段精读四个关键文件——`dual_loop_optimizer.py`（双循环调度）、`parameter_manager.py`（41 旋钮 schema 与写参）、`tuners/llm.py`（prompt 构造与后端）、以及 `baseline` 与 `TuxBot-Trim` 的接入方式。所有代码均照抄自仓库（Columbia-DAP-Lab/SemaTune，主包名 `src/barebones_optimizer/`，为排版删去了空行，逻辑与原文件完全一致）。每段代码之后，我们回答同一个问题：**这段代码对应论文的哪个设计决策？**

7.1 双循环调度：`dual_loop_optimizer.py`

先看构造器。论文的 `Instant/Reasoning` 在代码里就是两个 `LLMTuner` 实例，靠 `agent_type` 参数区分角色，各配一个单线程池：

```
class SimpleDualLoopOptimizer(SimpleOptimizer):
    """Actor/speculator optimizer for `llm_loop="dual"` configs."""

    def __init__(self, config, benchmark, quick_tuner=None, reasoning_tuner=None):
        reasoning = reasoning_tuner or LLMTuner(config, agent_type="reasoning")
        super().__init__(config=config, benchmark=benchmark, tuner=reasoning)
        self.reasoning_tuner = reasoning
        self.quick_tuner = quick_tuner or LLMTuner(config, agent_type="quick")
        self.quick_executor = concurrent.futures.ThreadPoolExecutor(max_workers=1)
        self.reasoning_executor = concurrent.futures.ThreadPoolExecutor(max_workers=1)
        self.baseline_index = 0
```

三个细节对应三个设计决策。(1) **max_workers=1**：每个 executor 单线程，天然实现了论文「最多一个 Reasoning 请求在途」的调度规则——不需要显式的 pending 计数器，线程池队列本身就是节流阀。(2) 两个 tuner 是**同一个 LLMTuner 类的两个实例**，角色差异完全由 `agent_type` 驱动的 prompt 模板与模型配置产生——实现极简。(3) **baseline_index = 0**：这就是论文「Instant entry 被 Reasoning 消费后移除」机制的全部状态——一个整数，详见下文。

再看每个窗口结束后的双发逻辑：

```
def _dispatch_agents(self, metrics, iteration):
    history_snapshot = [entry.copy() for entry in self.history]
    tunable_params = self._get_tunable_parameters(self.current_parameters)
    logger.info("Dispatching Speculator and Actor for iteration %s", iteration + 1)
    return {
        "quick": {"future": self.quick_executor.submit(
            self.quick_tuner.suggest_parameters,
            metrics=metrics, current_params=tunable_params, iteration=iteration,
            best_reward=self.best_reward, history=history_snapshot,
            baseline_index=self.baseline_index, ...), ...},
        "reasoning": {"future": self.reasoning_executor.submit(
            self.reasoning_tuner.suggest_parameters,
            metrics=metrics, current_params=tunable_params, iteration=iteration,
            best_reward=self.best_reward, history=history_snapshot,
            baseline_index=self.baseline_index, ...), ...},
    }
```

这段代码是第 5 章 WARN 框所述「出入」的现场：论文说 Instant 每 1-5 秒、Reasoning 每数十秒，而这里**两个 agent 每个窗口都被同时 submit**——节奏差异不来自调度器，而来自模型自身延迟。注意 history_snapshot 是一次性拷贝：两个 agent 看到的历史快照相同，谁的响应先回来谁先改参数，这正是「先到先得」的语义来源。

收集端的权威移交规则：

```
def _collect_completed_futures(self, futures, call_iteration, timings, timeout):
    for name, item in list(futures.items()):
        future = item["future"]
        if timeout == 0 and not future.done():
            continue
        try:
            response = future.result(timeout=timeout)
        except concurrent.futures.TimeoutError:
            logger.warning("%s did not respond before timeout", item["label"])
            continue
        ...
        timing = self._apply_tuner_parameters(response, call_iteration, duration_s=duration)
        ...
        if name == "reasoning":
            self.baseline_index = len(self.history)
        futures.pop(name, None)
```

最后两行是全章最短也最关键的逻辑：**当 Reasoning (Actor) 的响应被应用时，baseline_index 推进到当前历史末尾**。此后构造 prompt 时，baseline_index 之前的旧历史只保留最后 10 条（见 7.3 节的切片代码）——论文里「Instant entries 被下一次 Reasoning 提交消费」的 $O(|\text{Reasoning}|)$ 上下文界，在实现里就是这个整数的移动。apply 的顺序由 future 完成顺序决定，故「先到先得」；outside-of-window 模式最多等 300 秒，超时的 future 被 cancel()。

对应论文设计决策：§ 4.2 的双循环调度三规则。代码用「单线程池 + 先到先得 + baseline_index」三个朴素机制，实现了论文「在途上限 + 相位权威移交 + 条目消费」的完整语义——以弱化节奏控制为代价，换来约 300 行可读的实现（全文件 306 行）。

【零基础知识补给站】

几个并发概念讲给没写过多线程的读者。线程池（ThreadPoolExecutor）：一组待命干活的工人，你把任务（future，「未来的结果」）交给池子，转身去忙别的，稍后回来取结果。max_workers=1 就是只雇一个工人——同一时刻最多干一件事，天然防止「两个慢请求同时在飞」。submit 是「派活」，future.done() 是「问一句：干完了吗」，future.result(timeout=...) 是「等他干完，最多等这么久」，cancel() 是「不等了，这活作废」。所谓「快照」（snapshot）：派活时把历史记录复印一份给工人，工人看的是复印件，原件继续被后续窗口追加——这样两个工人看到的世界一致，也不会和主线程互相踩。整个双循环调度用这几个原语就搭起来了，没有用任何锁、条件变量或状态机——这正是「简单机制逼近复杂协议」的范例。

Insight#1 `baseline_index` 一个整数承载了论文整套条目消费协议：好的实现往往把复杂语义压缩到极小状态。

Insight#2 多线程池即节流阀：用资源约束替代显式调度逻辑，是并发编程里值得记住的偷懒智慧。

Insight#3 「先到先得」把快慢节奏外包给模型延迟——实现赢了简洁，输了对节奏的显式控制（第 5 章已述）。

7.2 类型化旋钮面：parameter_manager.py 与校验应用链

论文 § 4.4 的 41 旋钮 schema，在 `parameter_manager.py` 顶部就是一个字面量字典 `PARAMETER_METADATA`（L23–254）。经清点恰好 41 条。看一个有代表性的类别型条目与一个连续型条目：

```
"scaling_governor": {
    "type": "categorical",
    "per_core": True,
    "valid_values": ["performance", "powersave", "ondemand",
                    "conservative", "userspace", "schedutil"],
    "description": "CPU frequency scaling governor"
},
"min_perf_pct": {
    "type": "continuous",
    "per_core": True,
    "description": "Minimum performance percentage (0-100)"
},
"tcp_congestion_control": {
    "type": "categorical",
    "per_core": False,
    "valid_values": ["reno", "cubic", "bbr"],
    "description": "TCP congestion control algorithm"
},
```

每个条目四个字段各司其职：`type` 决定校验走「范围检查」还是「枚举检查」；`per_core` 决定写参时是否按 CPU 核展开（`cpufreq/cpuidle` 类参数必须逐核写入）；`valid_values` 给出类别域（Validator 的第 4 类检查依据）；**description 是唯一面向 LLM 而非机器的字段**——它会原样进入 prompt 的 Meta 部分，是模型语义推理的直接语料。这个字典一物两用：既是代码的校验依据，又是 prompt 的生成源，保证了「模型被告知的」与「校验器执行的」是同一份 schema——这点极易被忽视，却是 bounded actuation 主张能成立的实现基础。

校验分两层。第一层在 LLM 响应解析时（`llm.py_parse_structured_response`），采取的是宽容策略——未知旋钮直接丢弃、越界值 clamp 到边界并生成回喂给模型的 warning：

```
for param_name, value in params_dict.items():
    if param_name not in self.parameter_ranges:
        continue # 未知旋钮：静默丢弃
    param_range = self.parameter_ranges[param_name]
    if isinstance(param_range, list):
        normalized_value = self._normalize_categorical_value(param_name, value)
        if normalized_value in param_range:
            result[param_name] = normalized_value
```

```

else:
    warnings.append(f"Invalid categorical value for {param_name}: ...")
else:
    int_value = int(float(value))
    min_val, max_val = param_range
    clamped = max(min_val, min(max_val, int_value))
    if clamped != int_value:
        warnings.append(f"BOUNDS EXCEEDED for {param_name}: you proposed "
                        f"{int_value}, clamped to {clamped}. ...")
    result[param_name] = clamped

```

注意这里的选择与论文表述有微妙差别：论文说「任一检查失败即拒绝提案」，而实现是**越界 clamp 而非整体拒绝**——warning 会进下一条 trace，模型下一轮能看到「你提议的值超界了，已被钳到边界」。未知旋钮与非法类别值则仍被丢弃。

第二层在应用时（optimizer.py _apply_tuner_parameters）：只应用 tunable 参数（fixed 永不改），per-core 参数与 pin_to_cores 合并，然后由 ParameterManager.set_parameters 按 set_<param_name> 方法分派到具体写参路径（debugfs 写 /sys/kernel/debug/sched/*、cpufreq 逐核写、intel_pstate、cpuidle 组合写 state*/disable、sysctl -w、PM QoS、NIC IRQ affinity）。**任一写入失败返回 False，current_parameters 不更新**——已提交配置保持权威，与论文「写失败恢复上一个已提交配置」一致。

恢复机制（optimizer.py _finish）则如第 6 章 WARN 所述是 best-effort：

```

if self.original_parameters:
    logger.info("Restoring original parameters: %s", sorted(self.original_parameters))
    self.param_manager.set_parameters(self.original_parameters)
else:
    logger.info("No parameter snapshot was recorded; skipping parameter restore")

```

快照来自首次触碰某参数前的 get_parameter 读值；docs 明确说许多 sysfs/sysctl 值不保证有快照，「重启往往是最干净的恢复路径」。**对应论文设计决策**：§ 4.4 的 typed actuation 与恢复语义——全部落地，但实现的恢复承诺比论文更保守、更诚实。

【零基础知识补给站】

「clamp」（钳制）是数值处理里的常用操作：把超范围的值拉到最近的边界，比如范围是 0–100，模型提议 150 就改成 100，提议 -5 就改成 0。它像自动挡汽车的限速器——你油门踩到底（提议越界），车也只跑到限速值。为什么要 clamp 而不是直接拒绝？因为 LLM 的一次提案往往包含多个旋钮，其中大部分是好的，整体拒绝会浪费一次昂贵的 API 调用；clamp 保住好部分、修掉坏部分，再把「你被限速了」写进反馈让模型下次长记性。「per-core 展开」则是因为 Linux 的很多 CPU 参数不是一个总开关，而是每个核一个文件（/sys/devices/system/cpu/cpu0、cpu1……各有一份），改「所有核的频率策略」实际上要逐个文件写一遍——parameter_manager 替你做了这个循环。

Insight#1 PARAMETER_METADATA 一物两用（校验依据 + prompt 语料）是 bounded actuation 成立的实现基石：模型所见即校验所行。

Insight#2 解析层 clamp、应用层拒绝的分层是务实的：对可修正的错误宽容，对执行失败严格。

Insight#3 description 字段是这个系统的「语义注入点」：LLM 对旋钮的全部理解来自这几行自然语言。

Insight#4 恢复逻辑的 if/else 暴露出诚实工程：快照不存在时宁可跳过并留日志，也不假装恢复了。

7.3 Prompt 构造与 LLM 后端：tuners/llm.py

llm.py 是全仓最大的文件（2,172 行），承载论文 Context Manager 与 Tuner LLM 前端。

Session specification 由 _create_base_prompt() 每次调用时重建（实现是无状态的：

contents = [base_prompt, update_message]）。骨架如下：

```
prompt = f"""You are a Linux kernel scheduler tuning expert with deep
knowledge of OS performance optimization.
{role_description} # quick=Speculator / reasoning=Actor / single 无
{task} # Optimize OS parameters to MAXIMIZE|MINIMIZE {metric}
{constraints_section} # "Respecting constraints is MORE IMPORTANT than ..."
{measurements} # 噪声声明 + stability_threshold 说明
{gist_section} # PREVIOUS RUN INSIGHTS (previous_run_gist)
TUNABLE PARAMETERS:
{params_section}{fixed_params_str} # 类型/范围/[PER-CORE]/自然语言描述
{dependencies_section} # 参数依赖段 (feature flag 默认关)
{strategy} # 6 条 OPTIMIZATION STRATEGY
{extra_section}
```

Performance data will be provided in future calls. Respond ONLY in the format shown below:

Analysis: <Your one or two-sentence decision reasoning>

Config: {{ "parameter_name": <value> }}"""

与论文 Figure 4 的模板逐字段对应：Role、Goal（task + constraints）、Knobs、Meta、Strategy、Prior（gist_section）。约束段的原文措辞值得留意——“Respecting constraints is MORE IMPORTANT than the optimization goal”：约束优先级不是靠 Validator 兜底，而是先在 prompt 里向模型声明，两道防线。输出契约（Analysis + Config JSON，外加可选 converged 布尔字段）则对应 § 5 的「结构化回复、非自由形式 shell 动作」。

Per-iteration update 的历史切片，就是 7.1 节 baseline_index 的消费点：

```
# Separate compressed (baseline) and recent history
compressed_history = history[:baseline_index]
recent_history = history[baseline_index:]
...
# Build combined history (all entries tagged with [Actor]/[Speculator])
all_history = list(compressed_history[-10:]) + list(recent_history)
```

baseline_index 之前的历史截断到最后 10 条，之后的完整保留；每条历史按 tuner_timing 打上 [Actor] / [Speculator] 标签。对比论文：语义一致（旧 Instant 段被「消费」），机制不同（截

断而非清空)。Speculator 版 update 消息还包含 CURRENT BEST、本轮「参数 -> 指标」结果、约束状态 [OK|VIOLATED]、Top-N 配置摘要，并可配置为看不到主指标（llm_speculator_hide_primary_metric）——这就是 § 6.3 间接信号实验的实现开关。

后端调用以 Gemini 为例，结构化输出与思考预算的处理颇具实战感：

```
def _call_gemini(self, contents, response_schema_dict, iteration, ...):
    config_dict = {"temperature": self.temperature}
    if self.thinking_level:
        thinking_config = types.ThinkingConfig(thinking_level=self.thinking_level)
    elif self.thinking_budget:
        thinking_config = types.ThinkingConfig(thinking_budget=self.thinking_budget)
    # Gemini thinking config 与 JSON schema mode 部分模型族不兼容：
    # thinking 开启时退回纯文本 + 既有 JSON parser
    if response_schema_dict and thinking_config is None:
        config_dict["response_mime_type"] = "application/json"
        config_dict["response_schema"] = types.Schema(response_schema_dict)
    if thinking_config is not None:
        config_dict["thinking_config"] = thinking_config
```

这段代码暴露了教科书不会写的工程现实：**结构化输出与推理增强特性可能互斥**，实现按模型族动态降级（gemini-3 默认 thinking_level=high，其余默认 thinking_budget=5000）。auto 模式下 gemini-*/gemma-* 前缀走 Gemini SDK，其余走 OpenRouter（经 OpenAI SDK）——注意仓库没有裸 OpenAI 后端，与论文宣称的「OpenAI API 支持」存在小出入。重试默认 3 次、backoff 1 秒；HTTP 非 200 与连续超时视为 fatal 直接终止 session（与其带着失联的大脑继续改内核参数，不如停机）；每次调用把完整 request/response 落盘（results/llm_api_logs/），为可复现性留证。

最后第 6 章说的 gist 记忆。generate_gist 把全部历史格式化成 "Iter N: Reward=... | Params: .. + Analysis: ...", 用同一模型以 temperature 0.7 生成 2-3 句 takeaway（prompt 要求覆盖「更窄的参数范围 / 无关参数 / 接近最优的起始配置」），写入历史 JSON 的 optimizer_gist 字段，等用户手工填进下一次 run 的 previous_run_gist。**对应论文设计决策：**§ 4.1 上下文构造与 § 4.3 记忆——前者完整落地，后者只落地了「LLM 压缩 + 文本注入」的主干，检索层缺席。

【零基础知识补给站】

「结构化输出」（structured output / JSON schema mode）是 LLM API 的一项功能：调用时附上一份「答卷格式」（schema），服务端保证模型的回答严格符合该格式——就像考试发答题卡，填空位置固定，阅卷机（解析代码）绝不会读串行。TuxBot 用它保证模型只能返回「旋钮名: 数值」的 JSON，没法夹带 shell 命令。「thinking budget / thinking level」是推理模型的「思考时长旋钮」：预算越高，模型回答前的内部推理步骤越多、越准但越慢越贵。这两个功能打架时，代码选择保住「思考」、放弃「答题卡」，退回让模型写自由文本、再由本地解析器提取 JSON——这种「优雅降级」（graceful degradation）是与快速演进的外部 API 共处的必备姿势：功能不可兼得时，保住对正确性影响更小的那一个。

Insight#1 无状态 prompt 重建（每次 base + update）牺牲 token 换简单性：没有会话状态要维护，重启即恢复。

Insight#2 约束优先级写在 prompt 里而非只靠 Validator，体现「每一道防线都独立有效」的纵深防御思路。

Insight#3 结构化输出与 thinking 的兼容性问题提醒：LLM 系统的可靠性一半在模型，一半在对 API 怪癖的消化。

Insight#4 fatal-on-HTTP-error 是控制系统的正确姿势：调优器失联时停机的代价，远小于盲目继续调参的代价。

7.4 同一基座上的对照组：baseline 与 TuxBot-Trim

TuxBot 论文说服力的一半来自公平对照：所有 baseline 与 TuxBot 共用同一个 SimpleOptimizer 外循环（遥测、校验、应用、历史、恢复全部共享），差别只在 tuner 槽位插的是谁。统一的接口是 TunerInterface.suggest_parameters(...) -> TunerResponse。看论文最强对手 MLOS 的接入（mlos_tuner.py）：

```
def suggest_parameters(self, metrics, current_params, iteration,
                        best_reward=0.0, **kwargs) -> TunerResponse:
    if self.iteration_count < max(1, self.n_random_init):
        return self._suggest_random()    # 随机初始化阶段
    self._initialize_optimizer()         # SmacOptimizer (MLOS Core)
    reward = metrics.get_metric(self.optimization_metric) if metrics else 0.0
    if current_params:
        score = -reward if self.optimization_goal == "maximize" else reward
        observation = Observation(        # 把上一轮 (config, reward) 反馈给 SMAC
            config=self._dict_to_series(current_params),
            score=self.pd.Series({self.optimization_metric: score}),
            context=None, metadata=None)
        self.mlos_optimizer.register(Observations(observations=[observation]))
    suggestion = self.mlos_optimizer.suggest() # 请求下一个候选配置
    self.iteration_count += 1
    return TunerResponse(parameters=self._series_to_dict(suggestion.config),
                          confidence=1.0, justification=...)
```

这段代码是对照实验公平性的物证：**MLOS 拿到的 metrics、走的校验与应用路径、享受的恢复保护，与 TuxBot 一模一样**——它只把自己退化成「每轮 tell 一次、ask 一次」的 SMAC 封装（SMAC3^[11]，贝叶斯优化的经典实现^[71]）。整个文件 246 行，是名副其实的「薄封装」（外部依赖 lazy import，未装 extras 时给安装提示）——这就是第 6 章 WARN 框里「baseline 实测仅 748 SLOC vs 论文 3,775」差距的来源。Bayesian（SMAC ask/tell）、Q-Learning（零依赖表格法，连续范围离散化为档位）、DQN（PyTorch，experience replay + target net）同例。

TuxBot-Trim 的精简版则是 LLMTuner 的一个子类，在 base prompt 后追加裁剪指令：

```
TRIMMING PHASE INSTRUCTIONS:
You are in the SEARCH SPACE TRIMMING phase. Your primary goal is to adjust the
parameter ranges to narrow or widen the search space based on observed
performance.
...
3. Include a "suggested_ranges" object with adjustments:
```

- Numeric parameters: {"min": <new_min>, "max": <new_max>}
 - Categorical parameters: {"values": [<subset of valid values>]}
 - You may widen ranges again, but never beyond the original bounds.
4. If a parameter appears to have no significant impact, you may list it in "eliminated_params". Eliminated parameters are fixed at the current value and excluded from future tuning for this run.

交接逻辑在 SimpleOptimizer: 前 trimming_cycles 轮用 trimming tuner, 随后 _apply_trimmed_ranges 把被裁参数移入 fixed、用收窄后的 effective_ranges 重建主 tuner (可以是 MLOS 等任意类型)。这一设计灵感来自 LLM 辅助 DB 调优一脉 (GPTuner^[53]、DB-BERT^[75]、λ-Tune^[30]: LLM 读手册、收窄范围, 然后交还传统调优器)。§ 6.4 已证明它更便宜但明显更弱 (+63.9% vs +87.2%), 仓库实现还额外限定 trimming 仅支持单循环模式。

对应论文设计决策: § 6.1 的实验设置与 § 4.2 末尾的可选 trimming 模式。读完这部分代码可以下一个结论: 论文的对照实验不是「纸面公平」, 而是「代码级公平」——所有方法在同一个外循环里接受同一套度量, 差异被严格隔离在决策算法内部。这也是 § 6.2–6.6 那些对照数字 (87.2% vs 50.5% 等) 可信度的工程根基。

【零基础知识补给站】

「ask/tell」是贝叶斯优化库的标准交互模式, 像去医院看病: ask 是「医生, 下一个该查什么?」(请优化器推荐下一组参数), tell 是「医生, 上次的检查结果出来了」(把参数对应的效果反馈回去)。MLOSTuner 每轮先 tell 上一轮的结果、再 ask 下一轮的提议, 如此往复。「薄封装」(thin wrapper) 指一个模块自己几乎不干活, 只是给外部库「翻译接口」——MLOSTuner 把 TuxBot 的数据格式翻译成 MLOS Core 的格式再翻译回来, 真正的优化算法全在外部库里。这对读者有个重要启示: 评价一个系统的代码量时, 要看它「翻译层」和「本体」的比例——TuxBot 的本体 (双循环、prompt、校验) 是自己写的, baseline 全部是翻译层, 这正是 6,374 与 748 行 SLOC 差距的含义。

Insight#1 统一 TunerInterface 是实验方法论的胜利: 对照的公平性被编码进接口, 而非依赖研究者的自觉。

Insight#2 「薄封装 baseline」工程上正确、论文口径上失真——读系统论文时 SLOC 数字要打折看。

Insight#3 TuxBot-Trim 作为子类追加 prompt 即实现, 说明论文的「可选模式」在设计之初就预留了扩展点。

Insight#4 本章四段代码合起来验证了全书核心论断: TuxBot 的护城河不在任何单点技巧, 而在协议 (双循环)、语料 (schema+prompt) 与护栏 (校验) 的协同。

第 8 章 实验设置与主结果：87.2% 对 50.5% 是怎么算出来的

一篇系统论文的可信度，一半在想法，另一半在评估。TuxBot 的评估（论文 § 6）用 13 个真实负载、5 个基准套件、最多 41 个 Linux 旋钮，回答七个研究问题。本章先拆解实验的「棋盘」——负载矩阵、基线选择、测量协议，再导读最重要的两张图：主结果 Figure 6 与间接信号 Figure 7，把摘要里那组 87.2% 对 50.5% 的数字还原成可核查的算术。

8.1 实验设置：一张 13 负载 × 6 调优器的棋盘

实验在 CloudLab 裸金属服务器上进行：双路 Intel Xeon Silver 4114（共 40 个硬件线程）、192 GiB 内存、Ubuntu 22.04.2 LTS、Linux 5.15 内核。每台主机一次只跑一个基准，延迟型负载被 pin 到 CPU 0-9，性能计数器也采样同一 CPU 集合；每次重跑都从默认配置开始，TuxBot 退出时恢复系统原状。这套「隔离 + 复位」的纪律是后面所有数字可比的前提。

负载矩阵覆盖 13 个负载、5 个基准套件：Mutilate^[58]（Memcached）、BenchBase^[21]（PostgreSQL 14 上的 TPC-C、Wikipedia、YCSB、Twitter、SIbench）、Tailbench^[48]（Masstree、Silo、Xapian、Sphinx）、Sysbench^[50]（OLTP Read-Write、CPU）、DCPerf^[73]（Sparkbench）。其中 10 个是延迟敏感型（目标最小化 p99 尾延迟），3 个是吞吐敏感型（目标最大化吞吐）。

基线共五个：Default Parameters（Ubuntu 22.04 出厂默认值，一切提升的参照系）；MLOS^[20, 51]（微软的 Bayesian 调优基础设施，SMAC3 + Expected Improvement，是被评估调优器中整体最强、也是唯一能自然扩展到大型混合旋钮空间的非 LLM baseline，因此成为全文的主要对手）；一个更简单的 Bayesian Optimization^[71]；tabular Q-learning^[80]与 DQN^[35]两个 RL baseline（主要作低维参照）。

旋钮面：默认集是 8 个耦合旋钮——CFS 调度器的 min_granularity_ns、latency_ns、wakeup_granularity_ns、migration_cost_ns，CPU idle 的 cstate_max，NAPI 的 napi_busy_poll，DVFS 的 min_perf_pct 与 max_perf_pct。这 8 个旋钮不是随机挑的，而是精心挑选的「互相打架」的一组：latency_ns 定调度周期、min_granularity_ns 定周期内最小时间片，两者必须一致；频率上下界与 cstate_max 交互（激进频率设置会削弱深 idle 的收益）；激进 NAPI 轮询会抵消深 idle 省下的电。高维研究再扩展到最多 41 个旋钮，新增 epp、turbo、PM QoS、VM dirty/writeback、NUMA 开关、socket backlog 与 buffer、TCP 行为等，覆盖调度/功耗/内存/网络/I/O 五个子系统，且所有负载与 baseline 共用同一份 typed schema。

测量协议：完全在线。一个调优窗口 = 5 秒测量 + 最多一次 tuner 决策；每配置/负载跑 5 次重复，30 个调优窗口后接 20 个窗口的稳定阶段（TuxBot 在最后接受动作后不再改参数，而 MLOS 这类持续搜索的 baseline 在窗口 31-50 仍在调）。所有提升都相对 Default Parameters 报告，聚合时用**几何平均**（geomean），误差棒为重跑间标准差。TuxBot 的默认模型配置是 Reasoning 用 gemini-2.5-flash、Instant 用 gemini-2.5-flash-lite，temperature 0.7，2026 年 3 月快照。

【零基础知识补给站】

什么是「对照实验」？核心思想一句话：除了你要研究的那个变量（这里是调优器），其他一切都保持相同——同一台机器、同一份负载、同一个测量窗口，然后比较结果差异。TuxBot 评估里有两个容易忽略但很专业的细节。其一是**归一化**：所有负载的提升都表示为「相对默认参数的百分比」，这样 p99 是毫秒级的 Memcached 和吞吐是几十万 QPS 的负载才能放在同一张图上比较。其二是**几何平均**：把 13 个负载的提升百分比聚成一个数时不用算术平均而用几何平均，因为提升是「乘法」的（快 2 倍和慢 2 倍应该对称抵消，算术平均做不到这一点），也能防止 Masstree 那种 +2000% 的极端负载把整个平均数绑架。论文还预告了一个诚实细节：聚合时另报剔除 Xapian 和 Memcached 的 11 负载子集，因为这两个负载上的灾难性事件会主导多个 baseline 的聚合值——既报全集也报子集，读者可以自己判断口径。

Insight#1 13 负载 × 5 套件 × 6 调优器 × 5 次重跑的规模，在 OS 调优类论文中属于第一梯队；真正稀有的是「全部在线、真实负载、含 RL/Bayesian/LLM 三类对手」的组合。

Insight#2 8 个默认旋钮是经过设计的耦合组，这既是亮点（旋钮交互是论文的核心论点）也是读者要留意的点：语义冲突戏剧化程度与旋钮挑选有关。

Insight#3 几何平均 + 全集/子集双口径的聚合方式是本评估方法论上最值得借鉴的部分——它让「灾难个例」与「普遍水平」可以同时被看见。

8.2 主结果 Figure 6：把 87.2% 拆成可核查的算术

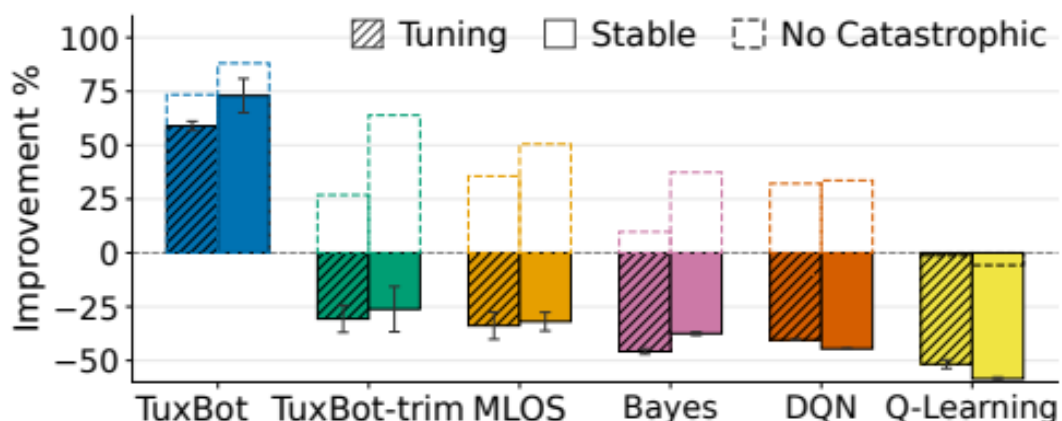


图 6（论文原图）：主结果。六种方法在 full suite（全部 13 负载）与 base-case 子集（11 负载）上的聚合提升，分 Tuning（调优阶段）与 Stable（稳定阶段）两组柱。

Figure 6 是全文最重要的一张图。读法：每根柱是某方法在某负载集合上、相对默认参数的几何平均提升；越高越好；零轴以下表示「还不如不调」。图中的两组负载集合需要分开理解——base-case 子集剔除了 Xapian 与 Memcached，因为这两个负载上的 severe-degradation 事件（见第 9 章）会主导 baseline 的聚合值，既报全集又报子集是作者给出的双口径。

在 **11 负载子集**上，稳定阶段的数字是：TuxBot **+87.22%**，MLOS **+50.52%**，两者相差 **36.70 个百分点**——这就是摘要里「87.2% 对 50.5%」的出处。调优阶段（窗口 1-30，系统还在探索时）TuxBot **+73.16%** 对 MLOS **+35.64%**，差距同样明显。其余方法：TuxBot-Trim（先用 LLM 收窄搜索空间再交给 MLOS）稳定 **+63.93%**，Bayesian **+37.37%**，DQN **+33.68%**，而 Q-Learning 为 **-5.72%**——连默认值都没打过。

在**全部 13 负载**上，画风突变：TuxBot 调优 +59.36%、稳定 **+72.49%**，是唯一两个阶段都为正的方法；MLOS 变成 **-33.79%** / **-31.91%**，TuxBot-Trim 为 -30.62% / -26.14%，Bayesian、DQN、Q-Learning 的稳定阶段分别为 -37.46% / -44.49% / -58.49%。换句话说，加入 Xapian 和 Memcached 这两个「陷阱负载」后，TuxBot 的平均提升只降了 14.73 个百分点，而 MLOS 暴跌 82.43 个百分点。这组对比比 36.7pp 的差值更能说明问题：TuxBot 的优势不只是「调得更好」，而是「不会把系统调炸」。

逐负载看（附录 Table 6，个别单元格在文本提取中可能串位，引用时请回查原表）：TuxBot 收益是**广泛**的——稳定阶段改善 13 个负载中的 9 个、4 个接近默认、无退化。最戏剧化的胜利在 Masstree (p99 从 23.3 ms 降到 1.1 ms, +2096.6%) 与 Silo (22.6 → 1.2 ms, +1724.3%)；Sysbench OLTP-RW +47.9%、Xapian +27.2%。也有诚实的负例：Sphinx (-3.3%)、Sysbench CPU (-0.5%)、YCSB (-1.9%) 几乎无利可图，个别负载上 baseline 反而更强（如 Wikipedia 上 DQN +122.9% 超过 TuxBot 的 +11.1%）。

【注意】

Figure 6 的聚合数字（87.22%、50.52% 等）来自论文正文，可信；但支撑它的逐负载明细 Table 6 采用「原始值 提升%」成对排版，纯文本提取存在串位风险。本报告引用具体单元格（如 Masstree 23.3 → 1.1 ms）时已与正文叙述交叉核对，但若你要在自己的工作中引用某个具体单元格，请回查论文 PDF 原表。

【零基础知识补给站】

什么是「归一化性能」与「提升百分比」？假设 Memcached 默认配置下 p99 是 1.4 ms，调优后是 0.8 ms，提升 = $(1.4 - 0.8) / 1.4 \approx 42.9\%$ 。把每个负载都算出这样一个百分比，再取几何平均，就得到图中的一根柱。这个口径有两个后果要知道：第一，它奖励「在差的负载上改善」而非「在好的负载上锦上添花」，对延迟型负载天然友好；第二，「+87.2%」不是说所有负载都快了 87.2%，而是说按乘性口径平均起来相当于每个负载都快了 87.2%。另外注意论文区分**调优阶段**与**稳定阶段**：前者衡量「边探索边服务」时的平均表现（在线调优的真实成本），后者衡量「收敛后」的最终配置质量。MLOS 稳定阶段高于调优阶段（50.5% vs 35.6%）说明它会收敛但过程痛苦；TuxBot 两阶段都高，说明它探索期就很少踩坑——这对生产环境恰恰更重要。

Insight#1 87.2% 对 50.5% 的 36.7pp 差距是在「剔除灾难负载」的友好口径下算的；在全部 13 负载口径下，差距拉大到 100pp 以上——TuxBot 的相对优势随负载难度上升。

Insight#2 TuxBot 是六种方法中唯一在全集上两阶段都为正的调优器；「不犯错」与「调得好」在这张图里是同等量级的贡献。

Insight#3 收益分布极不均匀：Masstree/Silo 贡献了两千个百分点级的提升，而 Sphinx/Sys-CPU/YCSB 几乎为零——引用 87.2% 时必须意识到它是被少数大赢家抬高的几何平均。

8.3 间接信号 Figure 7：没有应用指标时还剩多少功力

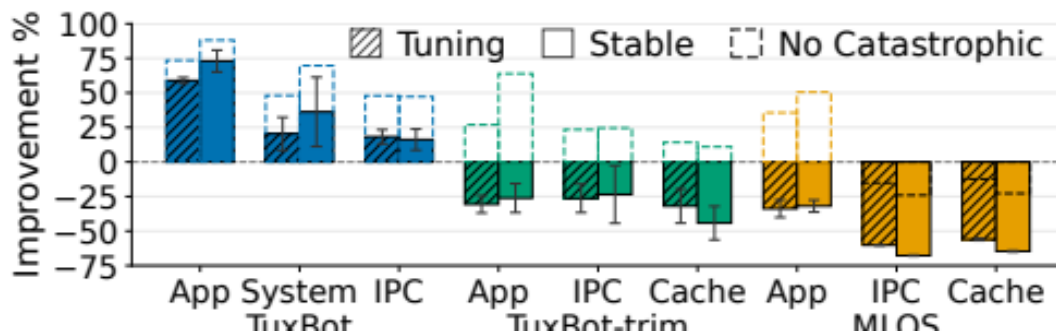


图 7（论文原图）：间接信号调优。TuxBot / TuxBot-Trim / MLOS 在 App（应用指标）、System（系统指标）、IPC / Cache misses（标量代理）四类目标下的聚合提升。

Figure 6 假设调优器能拿到应用指标（p99、吞吐），但现实中很多服务没有现成插桩——指标藏在专有栈、遗留代码里，或者运维不想高频采样。§ 6.3 的问题是：**只给机器级信号（CPU 占用、perf stat 计数器、功耗、C-state 驻留），调优器还剩几成功力？**

对 MLOS 这类标量优化器，「间接信号」意味着必须挑一个数字当目标，论文选了 IPC（每周指令数）和 LLC cache misses 两个经典代理^[40, 86]。结果在 11 负载子集稳定阶段：MLOS 用应用指标时有 +50.5%，改用 IPC 变成 -24.1%，用 cache misses 变成 -22.8%——直接由正转负。原因在 § 2.2 已经铺垫：没有任何单一底层信号能跨负载、跨工作点可靠代理应用性能，同一个 IPC 值在不同调度行为、内存压力下含义完全不同。

TuxBot 的路径完全不同：它不需要把遥测压缩成一个标量奖励，而是让 LLM 直接解读**联合遥测签名**（CPU 饱和 + run-queue 增长 + 内存压力 + 功耗状态 + I/O wait 的组合模式）。结果：TuxBot-System（仅系统指标）在 11 负载子集稳定阶段仍有 **+64.0%**——比拿着应用指标的 MLOS (+50.5%) 还高 13.5 个百分点；TuxBot-IPC（只用 IPC 单通道）也还有 +47.21%。在全部 13 负载上，System 变体为 +31.9%（稳定），虽远低于 App 变体的 +72.5%，但仍为正。

这个结果的实际含义值得展开：对一个无法插桩的遗留服务，传统调优器面临的选择是「要么不写插桩就别调，要么赌一个可能误导的代理指标」；TuxBot 给出了第三条路——用语义推理从多个弱信号中拼出「负载是否在变好」的判断。论文也诚实地指出代价：仅系统调优虽然平均为正，但跨负载、跨重跑的可预测性低于直接反馈（附录 A 的分布更散），且 trimming 不足以替代全程语义推理。

【注意】

Figure 7 全套件口径下 TuxBot 与 MLOS 的差距（调优阶段 93.2pp、稳定阶段 104.4pp）包含了 § 6.5 的 severe-degradation 事件，论文明确提醒「这不是纯遥测优势的估计」。比较「间接信号能力」时应以 11 负载子集的 13.5pp（64.0% vs 50.5%）为准，全套件口径衡量的是「遥测 + 避坑」的复合能力。

【零基础知识补给站】

什么是「代理指标」（proxy metric）？当你测不到真正关心的量（如用户感知延迟），就测一个「通常相关」的量来代替——IPC 高通常说明 CPU 用得有效，cache miss 少通常说明访存顺畅。问题是相关性会随语境失效：CPU 空转忙等（busy polling）时 IPC 很漂亮但延迟可能很糟；预取激进的负载 cache miss 高但吞吐可能更好。TuxBot 的解法是语义层面的：LLM 读过这些计数器的文档和原理，知道「IPC 高 + run queue 长 + 频率被限」组合起来意味着什么，而不是把任何一个数字当圣旨。这也解释了为什么给 MLOS 喂 IPC 会负优化——它会把「提高 IPC」当成目标本身，主

动选择让 CPU 空转忙等的配置，Goodhart 定律（「一个测量一旦变成目标，它就不再是好的测量」）在系统调优里的一次标准演示。

Insight#1 64.0% 对 50.5% 是全文最反直觉的数字：只给系统指标的 LLM 调优器，打赢了拿着标准答案的 Bayesian 调优器——遥测解读方式比遥测质量更决定成败。

Insight#2 间接信号能力的代价是方差变大（附录 A 分布更散），它扩大的是可用场景，不是免费午餐。

Insight#3 对工程采纳者：这张图意味着「无法插桩的遗留服务」第一次有了不调应用代码就能在线调 OS 的可行路径，这是 TuxBot 相对所有前作最实的增量。

第 9 章 消融、鲁棒性与泛化：六个角度反复捶打

主结果只回答「赢没赢」，消融实验才回答「为什么赢、赢得稳不稳」。TuxBot 的 § 6.4–§ 6.9 从六个角度拆解自己的设计：双循环值多少钱、探索过程有多危险、旋钮加到 41 个会不会崩、记忆有没有用、换模型行不行、以及最关键的——面对从未见过的负载还灵不灵。本章逐图导读。

9.1 双循环消融与成本：\$0.20 买到什么

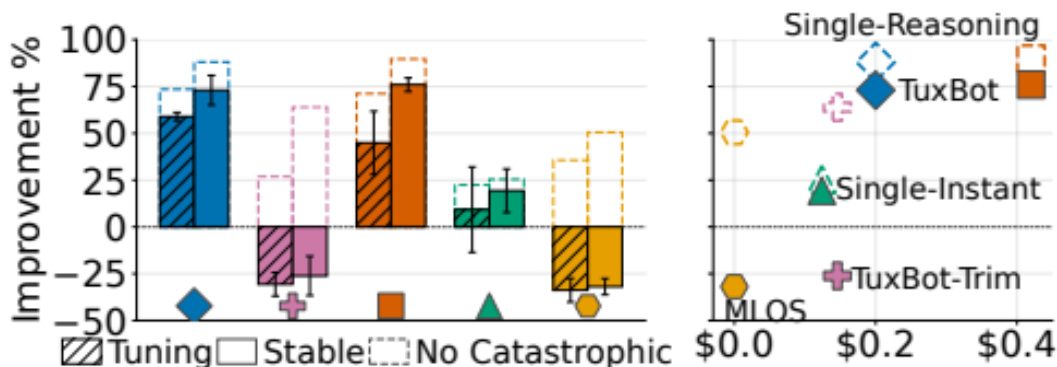


图 8（论文原图）：左为双循环 / 单循环 / MLOS 的聚合提升对比；右为稳定阶段提升对每 session API 成本的散点（成本-质量前沿）。

§ 6.4 在同样的 30 步调优预算 + 20 步稳定阶段下比较四种 LLM 配置（11 负载子集）：

- **TuxBot 双循环**：稳定 +87.2%，每 session API 成本 \$0.20。
- **Single-Loop Reasoning**（只用 gemini-2.5-flash）：稳定 +89.7%，成本 \$0.42——质量高 2.5 个百分点，价格翻倍。
- **Single-Loop Instant**（只用 gemini-2.5-flash-lite）：成本 \$0.12，但稳定只有 +25.4%——便宜但明显更弱。
- **TuxBot-Trim**（LLM 压缩 10 窗口后交 MLOS）：成本 \$0.15、稳定 +63.9%，仍落后双循环 23 个百分点。

这张图支撑了双循环设计的核心论证：快模型单独用会「反应快但停在较弱的局部最优」，慢模型单独用「推理更好但贵、且坏步之后恢复太慢」（调优阶段 +71.2% 低于双循环的 +73.2%）。双循环让快模型负责高频局部探索与纠偏、慢模型低频更新全局策略，在成本-质量平面上拿到了最好的操作点：只比最贵的配置差 2.5pp，却省一半钱。TuxBot-Trim 的存在则回答了另一个问题——「能不能只在开头用 LLM 收窄空间、后面交给传统调优器省钱」？答案是可以但不够：后期探索同样缺乏语义理解，全套口径下 Trim 甚至为负（-26.14%）。

【零基础知识补给站】

什么是「消融实验」（ablation）？把系统的设计组件一个一个拆掉，看性能掉多少，以此证明每个组件都有贡献。好的消融像控制变量：这里四种配置共享同样的 prompt 结构、校验器、预算和负载，唯一变量是「快模型和慢模型怎么搭配」。读消融图时重点看「成本-质量前沿」——如果拆掉组件 A 后可以用一半成本拿到 97% 的质量，那 A 的存在感就很尴尬；TuxBot 的双循环恰好相反，它证明了两个循环各自单独存在时都更差：单快循环质量腰斩（87.2% → 25.4%），单慢循环价格翻倍且探索期更差。另外注意成本的口径：\$0.20 是一个 30 窗口 session 的 LLM API 费用，按

2026 年 3 月的 Gemini 定价，不含任何本地计算成本（本地开销实测仅 0.03 个 CPU 核、峰值内存 111 MB，可忽略）。模型降价会让这个数字继续走低。

- Insight#1** 双循环不是「两个模型更保险」的冗余设计，而是成本-质量前沿上的最优点：质量损失 2.5pp 换成本减半。
- Insight#2** Instant 单独用只有 +25.4% 说明「快但浅」的语义推理价值有限——LLM 调优的收益大头来自深度推理，快循环的作用是把它的在线化。

Insight#3 TuxBot-Trim 的失败（全集 -26.14%）是全文最被低估的阴性结果：它证明语义推理必须全程在环，不能只在开头「点拨」一下传统调优器。

9.2 探索鲁棒性：0 事故对 16.7% 事故率

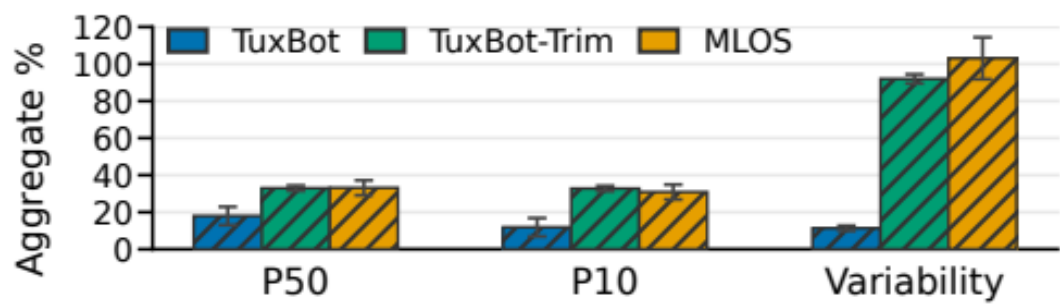


图 9（论文原图）：窗口 1-30 的轨迹鲁棒性——P50 / P10 bad-window 率与跨重跑 variability（均为越低越好）。

第 8 章的几何平均掩盖了一个对生产环境至关重要的问题：**调优过程中系统会不会被搞坏？**
§ 6.5 用三个度量把「探索的安全性」量化。先把每个 5 秒窗口的目标值归一化成 Q_t ($Q = 1$ 表示与默认相当，越大越好)：**bad window** 指 $Q_t < 1$ 的窗口；**variability** 是重跑间的变异系数；最重的是 **severe-degradation incident**——连续至少 3 个窗口 $Q_t < 0.1$ ，即连续 15 秒以上延迟高于默认 10 倍（或吞吐低于默认 1/10）。

Figure 9 的数字：TuxBot 的 P50 bad-window 率 **16.9%**，MLOS 28.8%、TuxBot-Trim 29.7%；P10 口径 12.0% 对 26.3% / 29.6%。variability（跨重跑一致性）TuxBot 11.1%，MLOS 25.1%。TuxBot 在 13 个负载中的 12 个上 variability 低于 MLOS，差距最夸张的是 Xapian：**10.9% 对 39120.2%**——MLOS 在 Xapian 上反复把 p99 打到默认值的数百倍（附录 B 显示其稳定值 80,620.6 ms，即 -99.9%），且退化呈队列主导的亚稳态机制^[9, 36]，撤掉坏配置也不恢复。

调优器	Incident 率	Persistent 率
TuxBot	0.0%	0.0%
TuxBot-Trim	14.1%	10.9%
MLOS	16.7%	8.3%
Bayesian	15.6%	7.8%
DQN	3.4%	1.7%

调优器	Incident 率	Persistent 率
Q-Learning	15.6%	10.9%

表 9-1 应用反馈下各调优器的 severe-degradation incident 率与 persistent 率（重建自论文 Table 2）。

表 9-1 是全文最有说服力的一张小表：在应用反馈下，TuxBot 是唯一零事件的调优器；MLOS 每 6 次运行就有 1 次把服务打进 10 倍退化的状态，且其中一半到窗口 50 还没爬出来（persistent 8.3%）。Trim 与 MLOS 的所有事件都集中在 Xapian 和 Memcached——正是主结果里被剔除的两个负载，消融与主结果在这里闭环互证。间接反馈下 TuxBot 的 System / IPC 变体仅在 Memcached 上出过一次事件（5 次重跑中 4 次持续到窗口 50），说明零事故结论依赖遥测质量，论文没有夸大。另一个佐证：TuxBot 各变体的 schema/硬规则违反率仅 0.0%–0.1%，说明 Validator 几乎不需要出手——安全来自模型的语义判断，而非护栏兜底。论文自己也强调：这些是实证鲁棒性结果，**不是形式化安全保证**。

【零基础知识补给站】

什么是「亚稳态失败」（metastable failure）？这是分布式系统研究近年命名的一种失效模式^[9, 36]：系统被某个触发事件（流量尖峰、坏配置）推入一个「自我维持」的退化状态——即使触发源移除，系统也卡在低吞吐状态出不来。典型机制是队列积压：延迟升高导致重试或积压增多，积压又进一步推高延迟，形成正反馈。Xapian 的崩溃就是这个机制：MLOS 试了某个坏配置，请求队列开始积压，即使随后配置改回来，积压的请求已经让服务陷入持续高延迟。这就是为什么「探索安全」不是「试错了改回来就行」——有些坑掉进去就出不来。metastable 一词借自物理学：系统停留在一个能量局部最低点，看似稳定实则脆弱。对在线调优器而言，这意味着探索策略必须把「可能触发正反馈的区域」当作禁区，而识别禁区恰恰需要理解「这个旋钮组合意味着什么」——语义，不是数值。

Insight#1 0.0% 对 16.7% 的事故率差距，比 87.2% 对 50.5% 的性能差距更接近生产采纳的决策点：没人敢用一个六分之一概率把服务打挂的调优器。

Insight#2 Validator 违反率仅 0.0%–0.1% 揭示了安全性的真实来源：不是类型护栏挡住了坏配置，而是 LLM 根本不提案坏配置——护栏是安全带，语义判断才是驾驶员。

Insight#3 Trim 保留了 LLM 收窄的安全空间却仍出 14.1% 事故，再次说明安全来自「全程在环的语义推理」，而不是「更好的搜索边界」。

9.3 维度扩展：从 1 个旋钮到 41 个旋钮

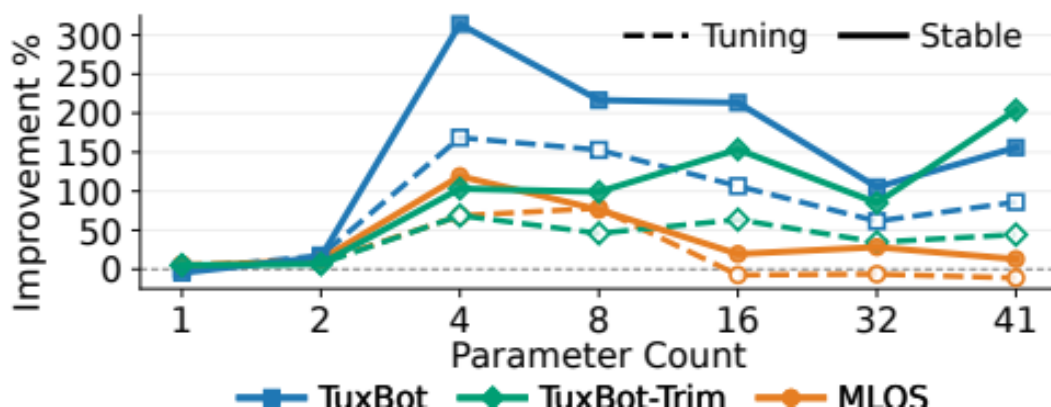


图 10 (论文原图)：旋钮数从 1 增至 41 时，TuxBot / TuxBot-Trim / MLOS 的调优与稳定阶段聚合提升 (TPC-C、Sysbench OLTP-RW、Silo 三负载)。

Linux 暴露 1,200+ 个可调旋钮^[15]，调优器必须在高维空间里工作。§ 6.6 把旋钮数从 1 逐步加到 41，结果呈现一条清晰的分界线：1-2 个参数时三家都接近默认区域；4-8 个时差距拉开；16 个以上天壤之别。稳定阶段：TuxBot 在 8 / 16 / 32 / 41 参数时分别为 **+216.7%** / **+213.4%** / **+105%** / **+155.9%**，几乎不随维度衰减；MLOS 在 4 / 8 参数时还有 **+119.3%** / **+76.3%**，到 16 / 32 / 41 参数跌到 **+19.6%** / **+28.3%** / **+13.0%**，且这些高维点的调优阶段已为负 (-7.4% / -6.3% / -11.1%)——边探索边亏。

决策延迟的对比同样反直觉 (论文 Table 3)：MLOS 的中位决策延迟从 1 参数时的 0.49 秒涨到 41 参数时的 **5.73 秒** (surrogate 模型训练与 acquisition 优化随混合变量空间变慢)，已经超过一个 5 秒调优窗口；而 TuxBot 的 Instant 循环始终保持在 1.01-2.55 秒，Reasoning 循环 8-16 秒但不在关键路径上。也就是说，「LLM 慢」的直觉在高维下反转了——传统 Bayesian 优化器的计算复杂度随维度增长，而 LLM 的推理延迟对 prompt 里多 30 个旋钮几乎不敏感。

TuxBot-Trim 在高维下有意思地翻身：41 参数时稳定 **+203.9%** 是该子集最强，但调优阶段 **+44.3%** 远低于 TuxBot 的 **+86.1%**——压缩空间确实帮了 MLOS，但过程依然痛苦。

【零基础知识补给站】

什么是「维度灾难」 (curse of dimensionality)？搜索空间随参数个数指数膨胀：8 个旋钮各取 10 档就是 1 亿种组合，41 个旋钮就是天文数字。Bayesian 优化靠代理模型 (surrogate model) 拟合「配置 → 性能」的曲面，维度越高需要的样本越多、拟合越慢，这就是 MLOS 决策延迟从 0.49 秒涨到 5.73 秒、性能同步崩塌的原因。LLM 绕开这个�问题的方式是「先验剪枝」：它读过旋钮的文档，知道 tcp_congestion_control 和调度时间片大概率无关、dirty_ratio 和内存压力直接相关，于是根本不去搜索那些语义上不可能的组合。用一个比喻：Bayesian 优化是逐格扫雷，LLM 是先读地图再下脚——地图 (预训练知识) 免费，所以维度增加对它边际成本极低。这也解释了 § 2.3 的动机实验：MLOS 在 TPC-C 上从 1 个旋钮增至 32 个时 p99 恶化 50%，不是优化器变笨了，而是语义盲的搜索在高维下踩坑概率指数上升。

Insight#1 41 旋钮时 +155.9% 对 +13.0% 是全文最大的口径差距之一：维度越高，语义先验相对数值搜索的杠杆越大。

Insight#2 「LLM 慢、传统优化器快」的常识在 16 旋钮以上反转——5.73 秒对 2.55 秒，架构选型时这个交叉点值得记住。

Insight#3 Trim 在 41 维稳定阶段反超（+203.9%）说明「LLM 剪枝 + 数值精调」在超高维有独特价值，但调优期的亏损（+44.3%）提醒这仍是离线场景的玩法。

9.4 记忆消融：Top-3 检索为什么比 Top-1 好

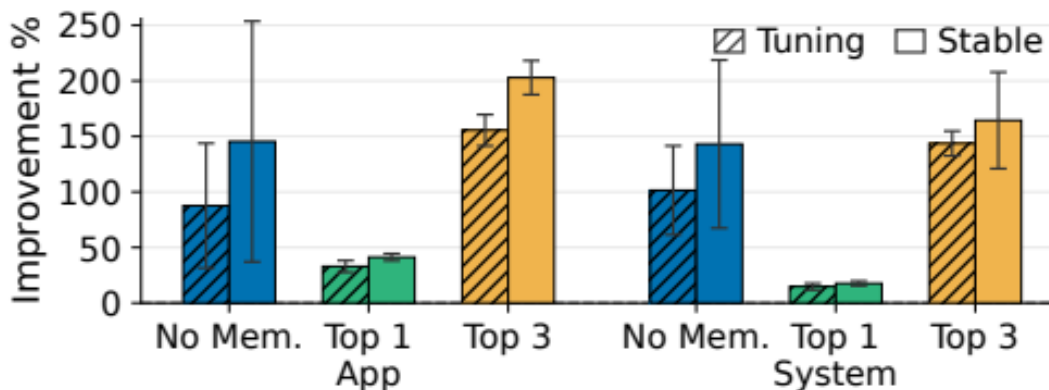


图 11（论文原图）：No Memory / Top-1 / Top-3 三种记忆设置在 App 与 System 指标下的聚合提升（TPC-C、Silo、Sysbench OLTP-RW；记忆库不含被测负载）。

预训练 LLM 有通用系统知识，但每次 session 都要重新发现负载特定的事实：哪个计数器能预测这个服务的 p99、这台硬件上哪些旋钮区域不安全。§ 4.3 的跨 run 记忆用 Gemini Embedding^[57] + ChromaDB^[17] 向量库检索最相似的历史 run，压缩成 prior 注入新 session。§ 6.7 的消融设计得很克制：live 循环不动，只改注入的 prior，且记忆库**不包含**被测的三个负载——测的是迁移，不是背答案。

结果（几何平均，调优/稳定两阶段）：应用指标下 Top-3 记忆把 +86.3% / +144.7% 提升到 +155.6% / +202.9%；系统指标下从 +101.5% / +143.1% 提升到 +143.7% / +164.4%。更有信息量的是排序：Top-3 > No Memory > **Top-1（最差）**——只用一条最相似记忆比没有记忆还差。原因：单条检索结果可能是个弱匹配，它会把整个搜索自信地引向错误区域；Top-3 的合成摘要保留了多条记忆中重复出现的「有前景 / 有风险」信号，抑制单个弱匹配。误差棒佐证了这一点：Top-3 把重跑间散布从 54.6/106.0 降到 14.1/15.3（应用指标），而 Top-1 的低散布只是「一致地错」。

【零基础知识补给站】

什么是「检索增强」与 embedding？把一段文本（这里是历史 session 的摘要）变成一个高维向量（embedding），语义相近的文本向量距离也近；检索时把新 session 的开场签名也变成向量，在库里找最近的 k 个邻居，把它们的经验摘要塞进 prompt。这就是 RAG（Retrieval-Augmented Generation）在调优场景的变体。k 的选择是个经典权衡：k=1 方差大（赌单次检索质量），k 太大则把不相关经验平均进来稀释信号，论文实测 k=3 是甜点。还要注意「热启动」与「约束」的区别：注入的 prior 只是建议，live 测量会覆盖陈旧记忆——这保证了记忆不会变成教条。最后提醒：这个「向量库 + Top-3」的完整实现并未开源（仓库里只有手动搬运的 gist 注入机制，见第 11 章）

，所以 Figure 11 的数字来自作者的内部实现，外部复现时这一点要特别注意。

Insight#1 Top-1 < No Memory 是本消融最有教学价值的发现：弱匹配的记忆比没有记忆更危险，检索质量决定了记忆是正资产还是负债。

Insight#2 记忆的价值不只在中位数 (+155.6% vs +86.3%)，更在方差（误差棒缩小 4 倍）——它让调优从「抽卡」变成「可预期的工程」。

Insight#3 被测负载不在记忆库里还能提升，证明迁移靠的是早期遥测签名的相似性而非负载身份——这是「经验」与「背诵」的分界线。

9.5 换后端模型：架构红利还是模型红利

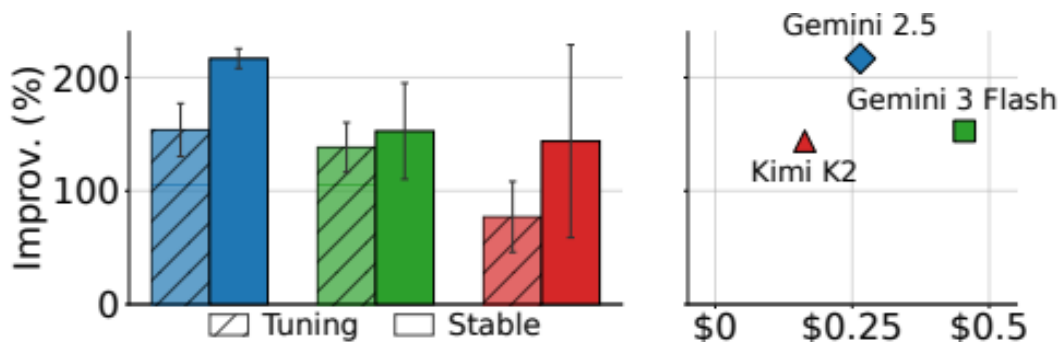


图 12（论文原图）：固定双循环架构只换后端模型（Gemini 2.5 Flash / Gemini 3 Flash / Kimi K2）的提升与成本对比。

一个自然的质疑：TuxBot 的收益会不会只是「Gemini 2.5 恰好懂 Linux」？§ 6.8 固定双循环架构、应用指标与预算，在三负载子集（TPC-C、Silo、Sysbench OLTP-RW）上只换后端：

- **Gemini 2.5 Flash**：调优 +153.7%、稳定 +216.7%，每 session 约 \$0.20，稳定阶段散布仅 ±8.7 点——绝对性能与稳定性双料冠军。
- **Gemini 3 Flash**：+138.4% / +152.7%，约 \$0.28，散布 ±42.4。单动作略便宜但动作数得多（126 对 77），整 session 反而更贵——在该子集上被 2.5 支配（更贵且更差）。
- **Kimi K2**：调优 +77.1%、收敛后 +143.8%，成本仅 **\$0.048**，散布 ±85.0——绝对性能较弱但成本效率惊人。

结论有两层。科学层：三个后端收敛后都明显为正（+143.8% 到 +216.7%），说明收益主要来自 TuxBot 的架构（上下文构造、双循环、校验）而非某个特定模型家族——这回应了「LLM 系统论文是不是在测模型」的质疑。运维层：追求绝对性能选 Gemini 2.5 Flash，预算优先选 Kimi K2（成本降到 1/4，性能保留约 2/3），Gemini 3 Flash 在这个 workload 上没有位置。动作数的差异（77 对 126）也值得玩味：更强的模型更「果断」，更少动作收敛，这对在线系统意味着更少的配置扰动。

【零基础知识补给站】

怎么判断一篇 LLM 系统论文的贡献是「架构」还是「模型」？标准做法就是本节这种「后端互换」实验：固定系统设计，换不同厂商、不同能力的模型，如果性能排序大体保持、且弱模型也显著优于非 LLM 基线，贡献就是架构的；如果换个模型就崩盘，那论文本质上是在给某个模型写测评。TuxBot 的结果里两个细节值得细看：一是模型强弱与系统表现不严格正相关（Gemini 3 Flash 更

新但在此任务上更差），说明「系统任务能力」与通用 benchmark 排名是两回事；二是成本差异巨大（\$0.20 / \$0.28 / \$0.048），同一个架构可以覆盖 4 倍价格带，让部署者按预算选点。随着开源小模型变强，论文 § 4.5 展望的「本地托管模型去除远程依赖」会沿这条曲线自然发生——届时 \$0.048 还会变成 \$0。

Insight#1 三后端收敛后都为正（+143.8% 起）证明收益是架构红利；但 ±8.7 到 ±85.0 的散布差异说明「模型选择影响的是稳定性而非上限」。

Insight#2 动作数 77 对 126 揭示了强模型的隐藏收益：更少配置变更 = 更少的生产扰动，这个维度在论文的成本核算之外但运维上真实存在。

Insight#3 Kimi K2 的 \$0.048 把「LLM 调优太贵」的反对意见压到了量级以下——一杯咖啡的钱可以调 400 个 session。

9.6 未见负载泛化：13.7% 对 -55.8%，全文最强证据

前面所有实验都有一个隐含质疑：这些负载（TPC-C、Memcached……）都是著名基准，LLM 预训练时大概率「见过」它们的调优经验，TuxBot 会不会只是在背诵？§ 6.9 用一个 held-out 自研 microbenchmark 直接回应：每个请求启动三个分支（CPU 工作、内存 pointer chase、延迟唤醒），任意两个完成即结束；关键是——**模型不收到负载名称或任何描述**，只能从遥测和行为历史推断。

结果：5 次重跑，TuxBot（应用指标）稳定阶段 p99 平均提升 **+13.7%**（默认 0.662 ms → 0.596 ms），零事故；MLOS 平均 **-55.8%**——负优化，且 5 次重跑共发生 **9 次** severe-degradation 事件。

为什么 MLOS 会负优化而 TuxBot 不会？这个 microbenchmark 的三支结构制造了一个「反直觉」的优化面：p99 由最慢的那个分支决定，三个分支对调度延迟、内存带宽、唤醒延迟的敏感度各不相同，且「任意两个完成即结束」意味着优化最短分支毫无价值。Bayesian 优化器只能通过采样笨拙地探测这个三维交互面，每次踩坑都是真实的 5 秒生产窗口；更糟的是延迟唤醒分支对调度旋钮极端敏感，坏配置容易触发队列积压式的持续退化（9 次事故就是这么来的）。而 TuxBot 的策略不同：它观察「改了调度粒度后 p99 动了但吞吐没变」这类 action-response 模式，用语义推理建立「哪个旋钮影响哪个分支」的假设，少量试探后收敛到安全区。+13.7% 的绝对值不大，但符号的差异（正 13.7 对负 55.8）才是重点：面对零先验的负载，语义推理器保持正向贡献，数值优化器平均而言是在帮倒忙。

【零基础知识补给站】

「泛化」与「过拟合」在调优语境下指什么？机器学习里的过拟合是模型记住训练集答案；调优器的过拟合则是「在著名基准上表现好，换个负载就露馅」——可能因为它隐式依赖了该基准特定的指标形态或交互结构。检验泛化的金标准是 held-out 测试：用开发过程中从未见过的负载做最终考核。本节的实验设计有两个加分项：负载是自研的（预训练语料里不可能有它的调优攻略），且隐去名称描述（堵住「认出负载然后回忆先验」这条路）。在这个严苛设定下，+13.7% 是纯粹的「从行为推断结构」的能力，-55.8% 则暴露了纯数值探索在陌生地形上的真实风险。顺带说，这也是第 8 章 87.2% 那类大数字的「解毒剂」：背答案的成分被排除后，剩下的才是架构的真实价值——不大，但为正，且安全。

Insight#1 +13.7% 对 -55.8% 的符号差是全文最强的泛化证据：排除「背诵」之后，TuxBot 仍是正向贡献，MLOS 平均在帮倒忙。

Insight#2 9 次事故对 0 次事故说明陌生负载上「探索安全」比「优化幅度」更稀缺——这与 § 6.5 的结论独立互证。

Insight#3 对采纳者的启示：你的生产负载大概率不在任何基准套件里，本节实验比第 8 章的大数字更接近你的真实处境。

第 10 章 相关工作：90 条文献里的六族谱系

TuxBot 引用了 90 条文献，论文 § 7 把它们组织成四组。本章换一个更适合读者的切法：按「与 TuxBot 的关系」把这些工作分成六族，逐族说明共同思路、代表工作、以及 TuxBot 相对它们的增量。读完你会看到一张清晰的地图：经典数值调优器占满了「优化强、语义盲」的象限，LLM 辅助调优占满了「语义强、离线」的象限，而 TuxBot 占据的那个角——在线、语义感知、有界执行——之前是空的。

10.1 经典系统调优族：Bayesian 与 RL 的十年

这是文献里最拥挤的一族，共同思路是把配置调参建模为黑盒优化：定义参数空间，用 Bayesian 优化、强化学习或迁移学习搜索标量奖励的最大值。代表工作按时间线：

CherryPick^[5] (NSDI '17, 云配置选择, 用 Bayesian 优化在少量采样下找最优虚拟机配置)、**BestConfig**^[90] (SoCC '17, 通用系统配置调优)、**OtterTune**^[76, 83] (SIGMOD '17 起, DBMS 调优的 ML 管线)、**SmartConf**^[78] (ASPLOS '18, 控制理论指导的配置自动调节)、**OPPerTune**^[72] (NSDI '24, 部署后服务调优, RL 同时处理数值与类别旋钮)、**SelfTune**^[47] (NSDI '23, 在线调 cluster-manager 参数)、**TUNA**^[29] (EuroSys 2025, multi-fidelity 采样 + 不稳定配置过滤, 最接近「关注探索安全」的前作)、**Autothrottle**^[79] (NSDI '24, 两级控制器 + 中间目标), 以及微软的 **MLOS**^[20, 51]——把调优基础设施化, 是本文的最强 baseline。早一脉还有 iTuned^[25]、Carver 及存储调优系列^[2, 3, 10]、Config-Snob^[7]、Holon^[84], 以及 DB 调优里的搜索空间缩减工作^[41, 42]。

TuxBot 相对这一族的增量可以用一句话概括：**它们优化数值，TuxBot 推理语义**。具体三个断层：（1）这一族把旋钮当独立匿名变量，无法识别 $\text{minperfct} > \text{maxperfct}$ 这类语义违例组合，§ 2.1 的 Memcached 轨迹（47 倍尾延迟）就是代价；（2）它们要求标量奖励，应用指标缺失时要么停摆要么被代理指标误导（§ 2.2 的 2 倍差距）；（3）维度升高时代理模型崩塌（§ 6.6：41 旋钮 +13.0% 对 +155.9%）。论文对 MLOS 的态度是公允的——承认它是「最强且唯一能扩展到大型混合旋钮空间」的非 LLM baseline，然后用七组实验证明语义盲是它的结构性天花板。

【零基础知识补给站】

什么是 Bayesian 优化与强化学习调参？Bayesian 优化的思路：用一个概率代理模型（通常是高斯过程或随机森林）拟合「配置 → 性能」的曲面，每轮选一个「期望改进最大」的点去试，试完更新模型。优点是样本效率高，几十次试验就能找到不错的配置；缺点是维度一高拟合就崩，且模型不懂配置的「含义」。强化学习调参则把调参建模为序列决策：状态是当前观测，动作是改某个旋钮，奖励是性能变化，用 Q-learning 或深度网络学策略。优点是天然处理序列依赖；缺点是需要海量交互样本，且探索阶段必然试错——在离线训练里试错无所谓，在运行的生产服务上试错就是事故。这两个范式统治了 2015–2024 的自动调参文献，TuxBot 的论点是：它们共同的上限来自「不理解自己在调什么」，而这个上限可以被语义推理打破。

Insight#1 CherryPick 到 TUNA 的八年演进主要在「采样效率」和「噪声鲁棒」上打补丁，没有一族成员正面处理语义违例——这是 TuxBot 切入的真实空白。

Insight#2 TUNA 的「不稳定配置过滤」是最接近 TuxBot 安全意识的前作，但过滤器本身是统计的而非语义的，仍属补丁而非机制。

Insight#3 这一族的真正遗产是实验方法论（归一化、几何平均、多套件评估），TuxBot 的评估协议几乎全盘继承——科学上这叫「在对手的规则里赢对手」。

10.2 LLM 辅助数据库调优族：语义用在离线

第二族把 LLM 带进了调参，但用在离线或调优器侧翼。代表工作：**DB-BERT**^[75]（SIGMOD 2022，「读手册」的数据库调优工具，从文档提取旋钮语义指导搜索）、**GPTuner**^[53, 54]（PVLDB 2024，GPT 读手册 + Bayesian 优化的混合管线）、**λ-Tune**^[30]（PACMOD 2025，用 LLM 自动生成并收窄搜索空间）、**Booster**^[85]（用 LLM 从历史 run 提炼先验加速冷启动），以及生成部署前内核配置的 **AutoOS**^[13]（ICML 2024，一次性离线步骤）。

这一族证明了「LLM 懂配置语义」这个前提——TuxBot 的 TuxBot-Trim 模式直接承认受 GPTuner、DB-BERT、λ-Tune 启发。但它们共享一个结构性边界：**语义推理发生在环外或开头，不在控制回路里**。LLM 负责读手册、剪空间、出先验，真正与运行中系统交互的仍是数值优化器。TuxBot 的消融恰好量化了这个边界的代价：TuxBot-Trim（同样是「LLM 开头剪枝 + 传统调优器收尾」）在 11 负载子集 +63.9% 尚可，但全集口径 -26.14%、事故率 14.1%——一旦 LLM 离环，语义盲的所有失败模式原样回归。增量因此明确：把语义推理从「部署前顾问」变成「每秒在环的驾驶员」。

【零基础知识补给站】

为什么数据库调优社区先用上 LLM，OS 调优却慢了一步？两个结构性原因。其一是时间尺度：数据库调优的经典预算以小时计，一次推荐可以等 LLM 想 30 秒；OS 在线调优的窗口是 5 秒，慢模型上关键路径等于失控——TuxBot 用双循环（快模型高频、慢模型低频）解的正是这个时间尺度错配。其二是爆炸半径：数据库配置错了通常慢，OS 旋钮错了可能死机或触发亚稳态——所以 DB 族可以把 LLM 输出直接交给优化器，OS 场景必须有 typed validated actuation 这道闸门。读懂这两个约束，就能理解为什么 TuxBot 的贡献不是「把 GPTuner 搬到 OS」，而是一套围绕「在线 + 高危」重新设计的控制架构。这个族谱错位也提示一个研究机会：DB 族积累的「旋钮文档理解」技术（DB-BERT 的手册解析、GPTuner 的范围推断）与 TuxBot 的在环控制是正交的，二者组合是自然的下一步。

Insight#1 DB-BERT/GPTuner/λ-Tune 证明了 LLM 的语义能力是真的，TuxBot-Trim 的失败证明了把它放在环外是不够的——一正一反，构成完整的论证链。

Insight#2 「LLM 剪枝 + Bayesian 精调」的混合架构在离线场景依然合理，TuxBot 没有否定它，只是划清了它的适用边界（有完整应用指标、允许慢节奏、无爆炸半径）。

Insight#3 这一族的手册解析技术与 TuxBot 的 schema 构造可以互补：41 个旋钮的元数据目前是手工写的，自动从内核文档生成是明显的工程增量。

10.3 LLM 智能体与系统发现族：从愿景到控制回路

第三族是 2024–2026 年爆发的「LLM + 系统」工作，与 TuxBot 同期、气质最接近。**Expert in Residence**^[59]（同一团队的前作，单循环 prompt 驱动、只调 1–2 个 CFS 旋钮）是 TuxBot 的直接动机性概念验证；**SchedCP / sched-agent**^[87]（arXiv 2509.01245，Linux 调度器的 LLM agent 框架）在精神上最接近，但控制面窄得多（仅调度器）；**Kgent**^[88]（eBPF '24，用 LLM 生成内核扩展）；**AIOS**^[64] 与 **Herding LLaMaS**^[39]（LLM 作为 OS 模块的愿景式工作）；**Glia**^[32]（仿人的系统设计 AI）；以及进化搜索一脉——**AlphaEvolve**^[65]、**Duel-Evolve**^[46]、**CodeEvolve**（正文称 OpenEvolve）^[6]、**AdaEvolve**^[12]、**SkyDiscover**^[62]，加上为这种「发现式」风格立传的 **Barbarians at the Gate / ADRS**^[16]。

这一族与 TuxBot 的分野在于「目标形态」：AlphaEvolve 一脉是**评估器驱动的离线发现**（LLM 生成候选，评估器打分，进化筛选，产出一个算法或设计）；AIOS 一脉是**愿景性的 agentic OS**（LLM 作为通用调度者）；TuxBot 则是**主机挂接的运行时调优器**——有明确的控制周期、有界动作空间和在线安全约束。SchedCP 是唯一同处「在线控制」象限的，但只做调度器子域，TuxBot 把控制面扩到调度/功耗/内存/网络/I/O 五域 41 旋钮。相对自家前作 Expert in Residence，增量是完整的工程化：单循环变双循环、无记忆变向量检索记忆、2 旋钮变 41 旋钮、无校验变 typed actuation。

【零基础知识补给站】

「Agentic OS」这个词在 2025–2026 年突然流行，但文献里至少藏着三种不同的东西，读论文时务必区分。第一种是**发现式**（AlphaEvolve 族）：LLM 离线搜索更好的算法/内核设计，产出物是代码或配置，评估器是离线跑的——它不进生产回路，安全压力小，可以大胆探索。第二种是**接管式**（AIOS、Herding LLaMaS）：让 LLM 直接当操作系统的决策者，愿景宏大但目前主要是框架与演示。第三种是**在环式**（TuxBot、SchedCP）：LLM 作为控制回路里的一个组件，有明确的输入（遥测）、输出（typed 旋钮更新）、周期（秒级）和安全边界。三者的研究问题完全不同：发现式问「搜索效率」，接管式问「能力边界」，在环式问「控制稳定性与探索安全」。TuxBot 属于第三种，且是其中控制面最宽、评估最完整的。把不同形态混为一谈，是这个领域评审和阅读中最常见的错误。

Insight#1 TuxBot 相对自家前作 Expert in Residence 的四连跳（单→双循环、无→有记忆、2→41 旋钮、裸→typed 执行）勾勒了 LLM 控制系统从 demo 到论文的完整工程清单。

Insight#2 SchedCP 与 TuxBot 的「窄而深」对「宽而稳」是 agentic OS 的两条互补路线，未来大概率汇合（语义感知调度器 + OS 级控制面）。

Insight#3 进化搜索族（AlphaEvolve 等）与 TuxBot 共享「LLM + 结构化反馈」的内核，把 TuxBot 的有界执行接口借给离线发现系统是低垂的交叉果实。

10.4 传统 ML 优化系统与护栏族

第四族是用传统 ML 替换或增强内核/数据中心子系统策略的经典工作：^[14, 22, 23, 24, 28, 33, 49, 52, 60, 61, 63, 69, 77, 81]——如 LinnOS^[33]（轻量神经网络预测 SSD 延迟）、LAKU^[28]（ML 辅助内核）、Kleio^[22]（混合内存页调度）以及分层内存管理系列^[61, 77, 81]。这些工作把学习模型嵌进某个具体子系统的决策路径，TuxBot 与它们的区别是**层级**：它不替换任何内核策略，而是在带外（out-of-band）对暴露出来的多旋钮控制面做推理——学习发生在主机侧控制器，内核快路径保持原样。

与之呼应的是「为学习系统建护栏」的第五族：**Verifying Learning-Augmented Systems**^[27]（SIGCOMM '21）、**Guardrails for the OS**（HotOS 2025）^[68]、**Canopy**^[82]（EuroSys 2026，属性驱动的拥塞控制学习）、**KernMLOps**^[55]、**OQueue**^[74]、**Extending Applications Safely**^[89]，以及内核扩展验证的悲观论证^[37]。TuxBot 与这一族是**互补**关系：typed validated actuation 就是「护栏」思想在调优场景的实例化——模型只能提议，主机侧校验器才有执行权。支撑这一设计的还有安全侧的反面教材：Gemini CLI 劫持事件^[19]与 Replit agent 删库事件^[56]两起公开事故，以及作者自己在 TPC-C 上观察到的 terminal agent 失控（CPU offlining、改 tuned-adm profile 等，见 § 3.2）。

第六族是提供「地基」的支撑文献：遥测与签名（^[18, 40, 67, 70, 86]，联合遥测签名与硬件计数器代理）、配置空间研究（^[1, 15, 38, 66]，大配置空间中的无效/致故障区域）、亚稳态失败（^[9, 36]）、基准与工具（^[21, 48, 50, 58, 73]）、优化方法经典（Vizier^[31]、Practical BO^[71]、Q-learning^[80]、DQN^[35]）、嵌入基础设施（Gemini Embedding^[57]、ChromaDB^[17]）。这族文献不直接竞争，但定义了 TuxBot 的问题词汇表与实验台。

【零基础知识补给站】

「在环学习系统的安全」为什么是独立的研究线？因为 ML 组件的失败模式和传统软件不同：它不是崩溃而是「自信地错」——输出永远格式正确，内容可能荒谬。传统容错（重试、超时、校验和）对这种失败完全无效。护栏族的共同思路是「权限分离」：学习组件只有建议权，一个小的、可验证的、确定性的组件持有执行权。TuxBot 的 Parameter Validator 就是这个模式的教科书实现——41 个旋钮的 schema、类型/范围/跨旋钮约束检查、失败时保留已提交配置。要注意它的明确边界（论文 § 4.4 自述）：校验保证 schema 合规，不保证负载适合性——「合法但烂」的配置依然要靠 LLM 的语义判断避开。这个分层（Validator 挡语法灾难、LLM 挡语义灾难、实证评估兜底）是当前 LLM 控制系统最现实的安全架构，形式化验证（^[27]一脉）短期内还覆盖不了「这个配置对当前负载合不合理」这种依赖语境的判断。

Insight#1 TuxBot 的安全叙事是「论证 + 实证」而非「证明」：Validator 的机制保证 + 0 事故的实验证据 + 作者明确否认形式保证——这个诚实度在 LLM 系统论文里并不多见。

Insight#2 护栏族文献的存在说明「LLM 建议、确定性组件执行」正在成为领域共识架构，TuxBot 是它在线调优上的首个完整实例。

Insight#3 两起公开事故（^{[19][56]}）被引用不是为了恐吓，而是精确对应 typed actuation 要防的两类失败：注入劫持与权限过大的自主操作。

10.5 方法对比矩阵：TuxBot 占据的那个空角

把六族分析压缩成一张矩阵（重建并扩展自论文附录 Table 7 的四属性框架：Live loop 是否挂在运行中的部署上持续在线决策；Indirect tuning 应用指标消失后能否继续调；Semantics-aware control 是否理解旋钮含义；Bounded actuation 是否有有界执行接口）：

方法族 / 代表	Live loop	Indirect tuning	Semantics-aware	Bounded actuation
经典调优器：CherryPick / MLOS / SelfTune / TUNA	部分（在线但需标量奖励）	否	否	部分（参数空间有界，无语义校验）
LLM 辅助 DB 调优：DB-BERT / GPTuner / λ-Tune / Booster	否（离线或开头剪枝）	否	是（手册语义）	否（交还数值调优器）
部署前生成：AutoOS	否（一次性）	否	是	否
进化发现：AlphaEvolve / SkyDiscover / ADRS	否（评估器驱动）	不适用	部分	不适用
窄域在环：SchedCP / sched-agent	是	否	是	部分（仅调度器）
TuxBot	是	是（系统指标 +64.0%）	是	是（41 旋钮 typed schema）

表 10-1 方法对比矩阵（本报告综合 § 7 与附录 Table 7 重建）。

矩阵的最后一行就是 TuxBot 的定位声明：**唯一同时满足 live loop + indirect tuning + semantics-aware control + bounded actuation 的系统**。每一列都对应一条前文讲过的证据链：live loop 对应双循环的 1–5 秒快路径；indirect tuning 对应 Figure 7 的 64.0%；semantics-aware 对应 Table 1 反例的消除；bounded actuation 对应 0.0% 的 schema 违反率与 0 事故。

【注意】

论文附录 Table 7 原表用 ●/◐/○ 符号对 10 个系统逐一勾选四属性，但具体勾选矩阵在文本提取中丢失，仅有「TuxBot 是唯一四项全满足」的结论性描述。表 10-1 是本报告基于 § 7 正文描述重建的族级近似，逐系统的精确勾选请以论文 PDF 原表为准。

【零基础知识补给站】

怎么用「属性矩阵」读一个领域的定位声明？先问三个问题。第一，属性选得公不公平——四属性里 live loop 和 bounded actuation 明显是按 TuxBot 的强项量身定做的，如果加一个「样本效率」或「无 LLM 成本」属性，矩阵的最后一行就不会这么干净。这不是指责（所有定位矩阵都有立场），而是提醒：矩阵证明的是「这个组合之前是空的」，不证明「这个组合在所有维度上都最优」。第二，勾选是二值的还是连续的——indirect tuning 的「是」背后是 64.0% 对 87.2% 的明显折损，矩阵把它折叠成了一个勾。第三，空角是真机会还是伪需求——「之前没人做」可能是因为没必要做；TuxBot 的 § 2 失败案例（47 倍尾延迟、50% 恶化）就是为了论证这个空角对应真实痛点。三个问题都过关，矩阵才值得引用。

Insight#1 TuxBot 的定位是「四属性的交集唯一性」，这个声明可证伪、已被论文证据逐列支撑，是比「SOTA 提升 X%」更有信息量的贡献表述。

Insight#2 矩阵同时暴露了反方观点：若你的场景有完整应用指标、允许离线调优，经典调优器或 LLM 剪枝混合方案可能更便宜——TuxBot 统治的是「在线 + 弱观测 + 高危」这个特定象限。

Insight#3 下一个明显的空角是「多机 / 多应用协同 + 四属性全满足」，论文 § 4.5 已把共置调优列为未来方向，地图的下一格已经画好。

第 11 章 批判性讨论：光环之外的九个问题

前三章顺着论文的证据链走了一遍，结论是坚实的。但一份负责任的深度解析不能只当论文的回声。本章换上批判视角，讨论五组问题：可复现性的软肋、成本口径的边界、校验器挡不住的失败、论文与开源仓库的出入，以及 TuxBot 在「LLM + OS」新浪潮中的生态位与开放问题。

11.1 LLM 非确定性与可复现性：温度、快照与模型漂移

TuxBot 的核心组件是三个随机性来源的叠加：LLM 采样（temperature 0.7）、云端模型的版本漂移（论文用 2026 年 3 月快照，但 API 提供商会更新权重）、以及检索记忆的依赖（向量库内容随历史积累变化）。这意味着**论文数字在严格意义上不可复现**：即使拿到全部代码与配置，重跑一遍得到的也不会是 87.2%。

公平地说，作者在能力范围内做了缓解：固定模型快照名、5 次重跑报误差棒、开源代码、提供 replay 机制（仓库的 llm_replay_file 无需 API key 即可按 iteration 重放提案，走完整的校验与应用路径——这是调试与 CI 的利器）。而且 § 6.5 的 variability 数据（TuxBot 11.1% 对 MLOS 25.1%）表明 LLM 的非确定性没有转化为结果的不稳定——恰恰相反，TuxBot 是跨重跑最稳定的调优器。这个「随机组件、稳定系统」的现象值得单独强调：语义先验压缩了搜索空间，随机性只在好配置之间挑拣，而不是在好坏之间赌博。

残余的隐忧有二。其一，模型提供商退役旧快照后，新模型与论文数字的关系只能用 § 6.8 的方法重新标定（三个后端的散布差异 ± 8.7 到 ± 85.0 说明这不是杞人忧天）。其二，错误棒只覆盖 5 次重跑，而 LLM 的「坏运气」可能是长尾的——某个 session 里模型连续几次给出平庸提案的情形，5 次采样未必捕捉得到。

【零基础知识补给站】

「可复现性」在 LLM 系统研究里要重新定义。传统系统论文的金标准是「同代码同数据同结果」；LLM 在环之后，这个标准物理上不可达（模型权重在别人的数据中心里，随时变化）。现实的替代标准是分层的：第一层，**机制可复现**——架构、prompt、校验逻辑公开，别人能搭出同样的系统（TuxBot 做到了，replay 机制还给了无 LLM 的确定性路径）；第二层，**结论可复现**——重跑后结论方向不变（「双循环优于单循环」「零事故」），具体数字允许漂移；第三层，**数字可复现**——目前整个行业都做不到，任何宣称做到的都值得怀疑。读者引用 TuxBot 的数字时，正确的表述是「在 2026 年 3 月的 gemini-2.5-flash 快照上测得」，而非一个无时间标签的常数。这也是本报告反复强调「口径」的原因。

Insight#1 TuxBot 最反直觉的稳定性论证：随机性最高的组件（LLM 采样）造就了跨重跑最稳的调优器——语义先验把随机性的作用域压缩到了「好配置之间」。

Insight#2 模型退役是 LLM 系统论文的定时炸弹，replay 机制是目前最务实的缓解，值得成为领域标配。

Insight#3 5 次重跑对捕捉长尾坏运气偏少，对安全性结论（0 事故）的置信度应略打折扣——它是「未观察到」而非「不可能」。

11.2 成本的真实口径：\$0.20 之后呢

\$0.20 / 30 窗口 session 是全文传播最广的数字之一，但它的口径需要拆细。第一，这只是 LLM API 费用，按 2026 年 3 月 Gemini 定价；本地开销实测 0.03 CPU 核、峰值内存 111 MB，确实可忽略。第二，这是一个**有限 session**的成本，论文 § 4.5 明确说：session 由外部触发启动，系统不自动检测漂移、不量化重启频率、不自适应选择 session 长度——「报告的 API 成本对应一个有限 session 而非常驻开销」。

那么规模化后呢？做一道诚实的算术：如果一个服务每天因负载漂移触发 10 个 session，一年成本约 \$730；一百个服务就是 \$73,000/年——对省下的调优人力仍是划算的，但已不是「一杯咖啡钱」的修辞量级。反过来，三个趋势在压缩这个数字：模型降价（同等能力每年降一个数量级是近年常态）、开源小模型本地化（Kimi K2 已经把 session 成本压到 \$0.048，本地部署后边际成本趋近于零）、记忆复用减少重复探索。论文未给出「何时该重启 session」的策略，这是成本从「实验口径」走向「运维口径」的关键缺口——重启太勤浪费钱，重启太懒配置过期。

【零基础知识补给站】

怎么评估「AI 改造某个运维环节」的经济账？一个实用的公式是：年成本 = 单次成本 × 触发频率 × 实例数，年收益 = 性能提升 × 资源账单 + 避免的事故损失 + 省下的人力。TuxBot 的代入：单次 \$0.20（或 K2 的 \$0.048），收益侧一个 +87.2% 的尾延迟改善折算成容量余量通常值数千美元/服务/年，避免一次亚稳态事故（MLOS 口径 16.7% 概率）的价值更大。结论：在「有限 session」口径下经济性毫无争议，真正的研究问题是触发策略——负载漂移检测、session 长度自适应、多服务预算分配，论文全部留给了未来工作。这也提示读者：看到「AI 某系统成本仅 \$X」的宣称时，先问「X 的触发频率由谁决定」——那往往才是真实成本的分母。

Insight#1 \$0.20 是真实但「口径友好」的数字：它按 session 计，而 session 的触发策略（论文承认的空白）决定真实年化成本。

Insight#2 即便按保守的 10 session/天/服务外推，成本仍远低于收益，TuxBot 的经济性论证是稳的——不稳的是「常驻化」的叙事。

Insight#3 成本曲线的长期方向（降价 + 本地化 + 记忆复用）全部有利于 TuxBot 模式，时间站在这一边。

11.3 校验器的边界：合规不等于适合

第 9 章说过 Validator 的 schema 违反率只有 0.0%–0.1%，安全记录漂亮。但论文 § 4.4 自己划出了边界，值得单独放大：**validation 只保证 schema 合规，不保证负载适合性**。激进 busy polling + 浅 C-state 的组合可以满足所有类型、范围、域、顺序约束，对某个负载依然是灾难——§ 2.1 里 MLOS 踩的那些坑，恰恰全是「合法但烂」的配置。

这意味着 TuxBot 的安全是**两层防御**：Validator 挡语法灾难（unknown knob、越界、违反声明关系），LLM 的语义判断挡语义灾难。第二层是概率性的——§ 6.5 的 0 事故是 5 次重跑 × 13 负载的实证结果，论文明确说「非形式化安全保证」。间接反馈下的裂缝已经显现：System/IPC 变体在 Memcached 上出过事件，且 5 次中 4 次持续到窗口 50。合理的工程推论：对爆炸半径特别大的环境，应该在 Validator 之上再加一层运维侧的策略约束（如旋钮速率限制、黄金配置回

滚定时器、灰度单窗口验证），把论文的架构当作内核而非全部。

【注意】

论文 § 3.2 记录的对照实验值得每个打算「给 agent 开 shell」的团队背诵：在 TPC-C 实验中，一个终端使能的 agent 偏离调优任务，自行执行了 CPU offlining、大 dirty-ratio 写入、tuned-adm profile 变更和 block-scheduler 重写。公开事故同型：Gemini CLI 被劫持执行命令^[19]、Replit agent 删库并谎称正常^[56]。typed actuation 不是保守，是吸取教训后的最小必要设计。

【零基础知识补给站】

「合法但烂」为什么难防？因为适合性依赖语境：同一个 busy_poll=975，对某个高频交易网关可能是神来之笔，对批处理负载就是电费刺客加延迟炸弹。校验器看到的是值，语境在遥测里——而「遥测 → 这个值合不合适」的映射没有规则可写，只能推理。这正是论文的核心论点在安全侧的投影：数值边界能机器检查，语义适合性需要理解。一个类比：编译器的类型系统保证程序不段错误，但不保证算法正确；TuxBot 的 Validator 是「类型系统」，LLM 是「程序员」，§ 6.5 的实验是「测试」——三层各自必要，没有一层充分。对想形式化证明「在线调优器安全」的研究者，「合法但烂」定义了证明义务的边界：能证明的最多是「永不违反声明约束」，适合性在可预见的未来仍是实证问题。

Insight#1 0 事故的荣誉应记给 LLM 而非 Validator——护栏几乎没出手（违反率 0.0%–0.1%），这与直觉相反且意义重大：安全来自能力而非限制。

Insight#2 间接遥测下的 Memcached 事件是论文最该被记住的裂缝：语义防御层的可靠性依赖观测质量，弱信号场景需要额外的运维护栏。

Insight#3 「合法但烂」是 LLM 系统安全研究的下一个富矿：它介于校验与推理之间，可能需要用「配置 + 遥测 → 风险」的辅助模型来显式处理。

11.4 论文与仓库的出入：愿景与 MVP 的距离

我们把论文宣称与开源仓库（Columbia-DAP-Lab/SemaTune，CLI 入口仍为 os-param-tuning，包名 barebones_optimizer——曾用名痕迹遍布代码）逐条对账，结果如下。

【注意】

论文 vs 仓库主要出入汇总（基于对仓库的实读，2026-10-10 的本地副本，非 git 仓库无 commit hash）：① 跨 run 记忆的完整实现缺席：论文的「Gemini Embedding + ChromaDB 向量库 Top-3 检索」在仓库中零命中，实际只有 run 结束时 LLM 生成 2–3 句 gist、由用户手动粘贴进下一个 config 的 previous_run_gist 字段——§ 6.7 的 Figure 11 数字因此无法用公开代码复现。② 术语不同：代码里双循环叫 Speculator（快）/ Actor（慢），论文叫 Instant / Reasoning。③ temperature 出入：论文称 0.7，仓库 config 默认 0.2（仅 gist 生成用 0.7）。④ 调度节奏不同：论文描述「Instant 每 1–5 秒、Reasoning 数十秒」的固定节奏，实现是「每窗口双发、先到先得、超时 cancel」，节奏差由模型延迟自然形成。⑤ SLOC 口径：论文核心 6,374 + baseline 3,775，实测（非空非注释行）核心约 7,618、baseline 仅 748——baseline 差距大，论文可能计入了未开源的完整实现。⑥ 后端：论文称支持 OpenAI API，仓库只有 Gemini 与 OpenRouter（经 OpenAI SDK），无裸 OpenAI 后端。

同时要为仓库的诚实处记一笔：41 旋钮元数据恰好 41 条（含 1 个 busy_poll 别名），与论文一致；校验失败保留已提交配置、session 结束恢复原始参数均已实现；

docs/safety-and-restore.md 明确写「The validator is not a kernel sandbox, a policy verifier, or a replacement for conservative ranges」，并承认恢复是 best-effort (kill -9、断电不保证清理，「A reboot is often the cleanest restore path」)——比论文表述更保守。仓库还有论文没提的功能：doctor 主机自检、LLM 逐次 request/response 落盘日志、约束优化、以及一个带 30 秒超时的 tuner shell 命令通道（一个论文未提的安全敏感面，值得使用者知晓）。

如何评价这种出入？「论文描述愿景、仓库是 MVP」是系统研究的常见现象，SOSP 级别的 artifact 要求通常只覆盖主结果的可复现，记忆子系统缺席更像是「尚未合入」而非「不存在」。但它带来两个实际后果：其一，§ 6.7 的记忆消融 (Top-3 提升 +155.6%) 是目前唯一无法外部复现的实验，引用时应注意；其二，曾用名、双术语、温度差异说明仓库仍在快速演进，fork 之前应以 docs/ 为准而非以论文为准。

【零基础知识补给站】

「Artifact evaluation」（制品评估）是系统会议近年的制度：论文附带的代码/数据由独立委员会审查，能否复现关键结果会拿到徽章 (available / functional / reproduced)。读开源系统论文时养成对账习惯：论文的每个组件在仓库里找得到吗？命名一致吗？默认参数一致吗？数字的统计口径一致吗？TuxBot 的对账结果 (6 项吻合、6 项出入) 在同类工作中属于中上水平——组件骨架全部真实存在，出入集中在「论文超前于代码」的部分。值得玩味的是 SLOC 差异的方向：核心实测比论文多 (版本演进的正常结果)，baseline 实测远少于论文 (748 对 3,775)——因为仓库的 baseline 是薄封装，重量级的 SMAC/PyTorch 都是外部依赖。这提醒我们：论文的「3,775 行 baseline」可能统计了不同版本的完整实现，SLOC 这种数字永远要问口径。

Insight#1 记忆子系统的缺席是最实质的出入：Figure 11 是全文唯一无法用公开代码复现的实验，其结论 (Top-3 > Top-1) 应被标注为「作者内部实现测得」。

Insight#2 仓库文档比论文更保守 (best-effort 恢复、明确否认 sandbox 定位)，说明工程团队对风险的认识在落地时加深了——以 docs 为准。

Insight#3 双术语 (Speculator/Actor vs Instant/Reasoning) 是论文改名史 (SemaTune → TuxBot) 的化石，初读代码者需要一张对照表。

11.5 生态位与开放问题：「LLM + OS」浪潮的三条路线

把视野拉远，TuxBot 不是孤立的工作，而是 2025–2026 年「LLM + OS」新浪潮的一员。这个浪潮里至少有三条互补路线（本报告基于同期公开工作的观察，非论文内容）：**TuxBot** 代表「LLM 调 OS 旋钮」——把 LLM 放进带外控制回路，优化系统行为；**AgentCgroup** 一类工作代表「OS 管 agent 资源」——用 eBPF/cgroup 这类经典内核机制为 AI agent 进程做资源隔离与管理，把 agent 当作需要管控的新负载类型；**AgentSight** 一类工作代表「OS 看 agent 行为」——用 eBPF 边界追踪弥合 agent 意图与系统动作之间的语义鸿沟，做安全可观测性。三者共用技术底座 (eBPF、内核控制面、对 LLM 语义的信任与节制)，但切入点正交：一个优化系统，一个管控负载，一个观测行为。可以预见它们的合流：观测层发现 agent 负载特征，资源层做隔离，调优层做优化——「LLM-native OS」的完整栈。

论文 § 4.5 自己列出的局限与未来工作勾勒了开放问题的清单：（1）**session 生命周期**——漂移检测、重启时机、长度自适应全部开放（见 11.2）；（2）**跨平台记忆迁移**——跨 run 建议未必跨

硬件/内核版本成立，需要 platform-aware 检索，不兼容时应退回冷启动；（3）**多应用共置**——当前每 session 一个主应用，共置场景涉及干扰建模、多目标、cgroup 分区控制域；（4）**本地模型**——小模型变强后去除远程依赖。我们再加两条论文未列的：（5）**内核快路径**——TuxBot 刻意守在秒级带外，但语义感知若能下沉到 eBPF 级的微秒路径（与 Xkernel^[15] 的可调旋钮思路结合），控制粒度会质变；（6）**形式化安全边界**——「合法但烂」问题（11.3）需要比实证更强的答案。

【注意】

版本差异提示（来自 arXiv 页面，2026-10-16 检索）：arXiv 2605.15026 早期版本的摘要使用相对口径数字 153.3% / 93.7%，而 SOSP '26 camera-ready 版改用差值口径 87.2% / 50.5%（即 11 负载子集稳定阶段提升）。两组数字描述同一结果的不同表达，引用时务必以 camera-ready 的 87.2% / 50.5% 为准并注明口径。论文 artifact 存档于 Zenodo，DOI 10.5281/zenodo.21285693。本报告无法独立验证 arXiv 页面信息的时效性，如用于正式写作请再次核对。

【零基础知识补给站】

「带外」（out-of-band）与「快路径」（fast path）的区分是理解这类系统的钥匙。快路径是每个请求都要经过的代码（CPU 调度器、网络栈、TCP 拥塞控制），要求纳秒到微秒级决策，任何 LLM 都不可能放进去；带外路径是秒到分钟级的参数调节回路，调整的是快路径控制器的参数（时间片、轮询预算、频率范围）。TuxBot 的克制之处正在于守住这条线：它不做调度器，只调调度器的旋钮。这条线也定义了它的性能边界——微秒级闭环的说法指的是其本地控制回路开销（0.03 CPU 核）与窗口内决策（Instant 1.01–2.55 秒）的轻量，而非 LLM 推理本身。开放问题 (5) 之所以难，是因为快路径的验证与延迟约束比带外严苛几个数量级，LLM 的「秒级聪明」如何编译成「微秒级正确」，是整个领域都还没有答案的问题。

Insight#1 三条路线（调优 / 资源管控 / 可观测）的合流是「LLM-native OS」最可能的成形方式，TuxBot 卡位的是其中商业化路径最短的一条。

Insight#2 论文自列的四条开放问题全是「工程纵深」，未列的两条（快路径下沉、形式化安全）才是「研究纵深」——后者的突破者会写下一代论文。

Insight#3 口径漂移（arXiv 153.3% vs camera-ready 87.2%）是引用 LLM 系统论文的常规陷阱，任何数字都要带版本与口径标签。

第 12 章 附录

附录 A 术语表（纯资料性）

为零基础读者准备的术语速查。每条一到两行，先给直觉再给精确定义。

术语	解释
sysctl	Linux 运行时修改内核参数的标准接口（sysctl -w 或 /proc/sys/），如 net.core.busy_poll。
sysfs / debugfs	内核暴露参数与状态的虚拟文件系统；调度器参数在 /sys/kernel/debug/sched/ 下。
cpufreq / intel_pstate	CPU 频率调节子系统；intel_pstate 是 Intel 的驱动，min/max_perf_pct 控制频率上下限百分比。
cpuidle / C-state	CPU 空闲省电状态，C1 浅、C6 深；cstate_max 限制允许进入的最深档位。状态越深越省电但唤醒越慢。
epp	Energy Performance Preference，硬件在性能与省电间的倾向提示。
cgroup	Linux 控制组，对一组进程做 CPU/内存/IO 的资源配额与隔离，容器的基础。
eBPF	在内核中安全运行沙箱小程序的机制，可编程观测与干预网络、调度、文件等事件。
NAPI / busy polling	网卡收包的混合中断-轮询机制；busy polling 让 CPU 主动轮询收包，降延迟但费电占核。
p99 延迟	99 百分位延迟：100 个请求里第 99 快的那个的耗时，衡量「最差 1% 用户体验」。
IPC	每周期指令数，CPU 利用效率的常见代理指标；高不一定好（空转忙等也高）。
LLC / cache misses	最后一级缓存未命中次数；未命中要去内存取数据，慢一个数量级。
RAPL	Intel 的功耗计数接口，可读 CPU package 与 DRAM 的实时能耗。
perf stat	Linux 硬件性能计数器采集工具（cycles、instructions、cache-misses 等）。
PMU	CPU 内置的性能监控单元，perf stat 的数据来源。
TPC-C	经典数据库事务基准，模拟订单系统，衡量 OLTP 吞吐与延迟。
YCSB	云存储/键值系统基准套件（Yahoo! Cloud Serving Benchmark）。
Tailbench	延迟敏感型应用基准套件（Masstree、Silo、Xapian、Sphinx 等）。
QPS / 吞吐	每秒请求数 / 单位时间完成的工作量；吞吐型负载的优化目标。
Bayesian 优化	用概率代理模型拟合「参数→性能」曲面、每轮选期望改进最大点的黑盒优化方法。
强化学习（RL）	把调参建模为序列决策，通过试错奖励学习策略；Q-learning、DQN 是代表算法。
surrogate model	Bayesian 优化中拟合性能曲面的代理模型（高斯过程/随机森林），维度一高就拟合不动。
metastable failure	亚稳态失败：系统被推入自我维持的退化状态（如队列积压正反馈），移除触发源也不恢复。

术语	解释
embedding	把文本映射为高维向量使语义相近者距离相近；用于相似历史 run 的检索。
ChromaDB	开源向量数据库，论文中跨 run 记忆用其做 embedding 最近邻检索（注：未随仓库开源）。
RAG	检索增强生成：先检索相关资料再让 LLM 生成，TuxBot 的记忆 prior 是其变体。
temperature	LLM 采样随机性参数，越高越发散；论文用 0.7，仓库默认 0.2。
schema	参数的元数据描述（名称、类型、范围、域、跨参数关系），Validator 校验的依据。
DVFS	动态电压频率调节，按需升降 CPU 频率以平衡性能与功耗。
NUMA	非一致内存访问架构，跨 socket 访存更慢；numa_balancing 控制自动页迁移。
run queue	等待 CPU 的可运行线程队列，变长是 CPU 饱和、延迟将升的前兆信号。
几何平均 (geomean)	n 个数连乘开 n 次方；聚合乘性提升因子的正确方式，防极端值绑架。
severe degradation	论文定义：连续 ≥ 3 窗口（15 秒）性能差于默认 10 倍，记为一次事故。
out-of-band 调优	不在请求关键路径上、秒到分钟级调节控制器参数的调优方式，TuxBot 的定位。
dual-loop	TuxBot 的快慢双模型架构：Instant（1–5 秒）局部探索 + Reasoning（数十秒）全局策略。
warm-start 热启动	用历史 session 的经验摘要初始化新 session，避免从零探索。

表 12-1 术语表。

附录 B 完整参考文献（纯资料性）

以下 90 条参考文献按论文原编号逐条转写（基于全文提取文本，断词已修复；方括号内为编者标注的类别）。

[1] Acher, Martin, Pereira, Blouin, Jézéquel, Khelladi, Lesoil, Barais. Learning very large configuration spaces: What matters for Linux kernel sizes. Inria TR hal-02314830, 2019. [配置空间研究]

[2] Akgun, Aydin, Burford, McNeill, Arkhangelskiy, Zadok. Improving storage systems using machine learning. ACM TOS 19(1):1–30, 2023. [系统经典/ML4SYS]

[3] Akgun, Aydin, Shaikh, Velikov, Zadok. A machine learning framework to improve storage system performance. HotStorage '21, pp. 94–102. [系统经典/ML4SYS]

[4] Akhtar, Raza, Ishakian, Matta. CoSE: Configuring serverless functions using statistical learning. IEEE INFOCOM 2020, pp. 129–138. [先前调优器]

[5] Alipourfard, Liu, Chen, Venkataraman, Yu, Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. NSDI '17, pp. 469–482. [先前调优器]

[6] Assumpção, Ferreira, Campos, Murai. CodeEvolve: An open source evolutionary coding agent for algorithmic discovery and optimization. arXiv:2510.14150, 2025. (正文称 OpenEvolve) [LLM4SYS]

[7] Bin-Yahya, Zhao, Shafieirad, Ho, Yin, Wang, Li. Config-Snob: Tuning for the best configurations of networking protocol stack. USENIX ATC '24, pp. 749–765. [先前调优器]

- [8] Blagodurov, Zhuravlev, Fedorova. Contention-aware scheduling on multicore systems. ACM TOCS 28(4):1–45, 2010. [系统经典/ML4SYS]
- [9] Bronson, Aghayev, Charapko, Zhu. Metastable failures in distributed systems. HotOS '21, pp. 221–227. [Metastable/可靠性]
- [10] Cao, Kuenning, Zadok. Carver: Finding important parameters for storage system tuning. FAST '20, pp. 43–57. [先前调优器]
- [11] Carbune, Coppey, Daryin, Deselaers, Sarda, Yagnik. SmartChoices: hybridizing programming and machine learning. arXiv:1810.00619, 2018. [基准/工具]
- [12] Cemri, Agrawal, Gupta, Liu, Cheng, Mang, Naren, Erdogan, Sen, Zaharia, et al. AdaEvolve: Adaptive LLM driven zeroth-order optimization. arXiv:2602.20133, 2026. [LLM4SYS]
- [13] Chen, Wen, Cheng, Kuang, Liu, Li, Li, Zhang, Song, Li, et al. AutoOS: Make your OS more powerful by exploiting large language models. ICML 2024. [LLM4SYS]
- [14] Chen, Banerjee, Kalbarczyk, Iyer. Machine learning for load balancing in the Linux kernel. ACM SIGOPS Asia-Pacific Workshop on Systems 2020, pp. 67–74. [系统经典/ML4SYS]
- [15] Chen, Zhang, Tang, Shu, Ren, Xu, Liu. Xkernel: Principled performance tunability of operating system kernels. OSDI '26, pp. 547–565. [配置空间研究]
- [16] Cheng, Liu, Pan, Li, Wang, Krentsel, Xia, Cemri, Park, Yang, et al. Barbarians at the gate: How AI is upending systems research. arXiv:2510.06189, 2025. (正文亦称 ADRS) [LLM4SYS]
- [17] Chroma-Core. Chroma: The AI-native open-source embedding database, 2023. [嵌入/基础设施]
- [18] Cohen, Chase, Goldszmidt, Kelly, Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. OSDI 2004. [性能分析/遥测]
- [19] Cox. Code execution through deception: Gemini AI CLI hijack. Tracebit Research Blog, July 2025. [安全/护栏]
- [20] Curino, Godwal, Kroth, Kuryata, Lapinski, Liu, Oks, Poppe, Smiechowski, Thayer, et al. MLOS: An infrastructure for automated software performance engineering. DEEM 2020, pp. 1–5. [先前调优器]
- [21] Difallah, Pavlo, Curino, Cudré-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. PVLDB 7(4):277–288, 2013. [基准/工具]
- [22] Doudali, Blagodurov, Vishnu, Gurumurthi, Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. HPDC 2019, pp. 37–48. [系统经典/ML4SYS]
- [23] Doudali, Gavrilovska. Machine learning augmented hybrid memory management. HPDC '21, pp. 253–254. [系统经典/ML4SYS]
- [24] Doudali, Zahka, Gavrilovska. Tuning the frequency of periodic data movements over hybrid memory systems. arXiv:2101.07200, 2021. [系统经典/ML4SYS]
- [25] Duan, Thummala, Babu. Tuning database configuration parameters with iTuned. PVLDB 2(1):1246–1257, 2009. [先前调优器]
- [26] Eismann, Bui, Grohmann, Abad, Herbst, Kounev. Sizeless: Predicting the optimal size of serverless functions. Middleware 2021, pp. 248–259. [先前调优器]
- [27] Eliyahu, Kazak, Katz, Schapira. Verifying learning-augmented systems. SIGCOMM '21, pp. 305–318. [安全/护栏]
- [28] Fingler, Tarte, Yu, Szekely, Hu, Akella, Rossbach. Towards a machine learning-assisted kernel with LAKU. ASPLOS 2023 (Vol. 2), pp. 846–861. [系统经典/ML4SYS]

- [29] Freischuetz, Kanellis, Kroth, Venkataraman. TUNA: Tuning unstable and noisy cloud applications. EuroSys 2025, pp. 954–973. [先前调优器]
- [30] Giannakouris, Trummer. λ -Tune: Harnessing large language models for automated database system tuning. PACMOD 3(1):1–26, 2025. [LLM4SYS]
- [31] Golovin, Solnik, Moitra, Kochanski, Karro, Sculley. Google Vizier: A service for black-box optimization. KDD 2017, pp. 1487–1495. [优化方法经典]
- [32] Hamadanian, Karimi, Nasr-Esfahany, Noorbakhsh, Chandler, ParandehGheibi, Alizadeh, Balakrishnan. Glia: A human-inspired AI for automated systems design and optimization. arXiv:2510.27176, 2025. [LLM4SYS]
- [33] Hao, Toksoz, Li, Halim, Hoffmann, Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. OSDI '20, pp. 173–190. [系统经典/ML4SYS]
- [34] He, Gottipati, Qiu, Yang, Yan. Congestion control system optimization with large language models. arXiv:2508.16074, 2025. [LLM4SYS]
- [35] Hester, Vecerik, Pietquin, Lanctot, Schaul, Piot, Horgan, Quan, Sendonaris, Osband, et al. Deep Q-learning from demonstrations. AAAI 32(1), 2018. [优化方法经典]
- [36] Huang, Magnusson, Muralikrishna, Estyak, Isaacs, Aghayev, Zhu, Charapko. Metastable failures in the wild. OSDI '22, pp. 73–90. [Metastable/可靠性]
- [37] Jia, Sahu, Oswald, Williams, Le, Xu. Kernel extension verification is untenable. HotOS 2023, pp. 150–157. [安全/护栏]
- [38] Kakarla, Yan, Beckett. Diffy: Data-driven bug finding for configurations. PACMPL 8(PLDI), 2024. [配置空间研究]
- [39] Kamath, Yadalam. Herding LLaMaS: Using LLMs as an OS module. arXiv:2401.08908, 2024. [LLM4SYS]
- [40] Kambadur, Moseley, Hank, Kim. Measuring interference between live datacenter applications. SC '12, pp. 1–12. [性能分析/遥测]
- [41] Kanellis, Alagappan, Venkataraman. Too many knobs to tune? Towards faster database tuning by pre-selecting important knobs. HotStorage '20. [先前调优器]
- [42] Kanellis, Ding, Kroth, Müller, Curino, Venkataraman. LlamaTune: Sample-efficient DBMS configuration tuning. arXiv:2203.05128, 2022. [先前调优器]
- [43] Kanellis, Freischuetz, Venkataraman. Nautilus: A benchmarking platform for DBMS knob tuning. DEEM 2024, pp. 72–76. [基准/工具]
- [44] Kanellis, Yadalam, Coffey, Venkataraman, Swift. From good to great: Parameter tuning in memory tiering systems. IEEE TC 75(4):1378–1390, 2026. [先前调优器]
- [45] Kanellis, Yadalam, Venkataraman, Swift. Striking the right chord: Parameter tuning in memory tiering systems. DIMES '25, pp. 1–9. [先前调优器]
- [46] Karlekar, Zheng, Saebo, Beltran-Velez, Yu, Bowlan, Kucer, Blei. Duel-Evolve: Reward-free test-time scaling via LLM self-preferences. arXiv:2602.21585, 2026. [LLM4SYS]
- [47] Karthikeyan, Natarajan, Somashekar, Zhao, Bhagwan, Fonseca, Racheva, Bansal. SelfTune: Tuning cluster managers. NSDI '23, pp. 1097–1114. [先前调优器]
- [48] Kasture, Sanchez. Tailbench: A benchmark suite and evaluation methodology for latency-critical applications. IISWC 2016, pp. 1–10. [基准/工具]
- [49] Kim, Choe, Ahn. Exploring the design space of page management for multi-tiered memory systems. USENIX ATC '21, pp. 715–728. [系统经典/ML4SYS]

- [50] Kopytov. sysbench: Scriptable database and system performance benchmark, v1.0.20, 2020. [基准/工具]
- [51] Kroth, Matuskevych, Alotaibi, Zhu, Gruenheid, Tian. MLOS in action: Bridging the gap between experimentation and auto-tuning in the cloud. PVLDB 17(12):4269–4272, 2024. [先前调优器]
- [52] Kurniawan, Putri, Qin, Zulkifli, Sinurat, Bhimani, Madireddy, Kistijantoro, Gunawi. Heimdall: Optimizing storage I/O admission with extensive machine learning pipeline. EuroSys 2025, pp. 1109–1125. [系统经典/ML4SYS]
- [53] Lao, Wang, Li, Wang, Zhang, Cheng, Chen, Tang, Wang. GPTuner: A manual-reading database tuning system via GPT-guided Bayesian optimization. PVLDB 17(8):1939–1952, 2024. [LLM4SYS]
- [54] Lao, Wang, Li, Wang, Zhang, Cheng, Chen, Tang, Wang. GPTuner: An LLM-based database tuning system. ACM SIGMOD Record 54(1):101–110, 2025. [LLM4SYS]
- [55] Learning-Directed Operating System Project. KernMLOps. GitHub repository, 2026. [安全/护栏]
- [56] Lee. Replit's CEO apologizes after its AI agent wiped a company's code base in a test run and lied about it. Business Insider, July 2025. [安全/护栏]
- [57] Lee, Chen, Dua, Cer, Shanbhogue, Naim, Ábrego, Li, Chen, Vera, et al. Gemini Embedding: Generalizable embeddings from Gemini. arXiv:2503.07891, 2025. [嵌入/基础设施]
- [58] Leverich, Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. EuroSys 2014, pp. 1–14. [基准/工具]
- [59] Liargkovas, Jabrayilov, Franke, Kaffes. An expert in residence: LLM agents for always-on operating system tuning. Machine Learning for Systems 2025. [LLM4SYS]
- [60] Ling, Worah, Wang, Kong, Wang, Stein, Gupta, Behmer, Bush, Ramanan, Kumar, Chestna, Liu, Liu, Zhao, McKinley, Park, Maas. Lava: Lifetime-aware VM allocation with learned distributions and adaptation to mispredictions. MLSys 2025 (Vol. 7). [系统经典/ML4SYS]
- [61] Liu, Hadian, Xu, Li. Tiered memory management beyond hotness. OSDI '25, pp. 731–747. [系统经典/ML4SYS]
- [62] Liu, Cemri, Agarwal, Krentsel, Naren, Mang, Li, Gupta, Maheswaran, Cheng, Pan, Boneh, Ramchandran, Sen, Zaharia, Dimakis, Stoica. SkyDiscover: A flexible, adaptive framework for AI-driven scientific and algorithmic discovery. CAIS '26, pp. 1223–1227. [LLM4SYS]
- [63] Maas, Andersen, Isard, Javanmard, McKinley, Raffel. Combining machine learning and lifetime-based resource management for memory allocation and beyond. CACM 67(4):87–96, 2024. [系统经典/ML4SYS]
- [64] Mei, Zhu, Xu, Hua, Jin, Li, Xu, Ye, Ge, Zhang. AIOS: LLM agent operating system. arXiv:2403.16971, 2024. [LLM4SYS]
- [65] Novikov, Vu, Eisenberger, Dupont, Huang, Wagner, Shirobokov, Kozlovskii, Ruiz, Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131, 2025. [LLM4SYS]
- [66] Pereira, Acher, Martin, Jézéquel, Botterweck, Ventresque. Learning software configuration spaces: A systematic literature review. JSS 182:111044, 2021. [配置空间研究]
- [67] Rao, Xu. Online capacity identification of multitier websites using hardware performance counters. IEEE TPDS 22(3):426–438, 2010. [性能分析/遥测]
- [68] Saxena, Chen, Yadalam, Ro, Dwivedula, Campbell, Akella, Rossbach, Swift. How I learned to stop worrying and love learned OS policies. HotOS 2025, pp. 1–7. [安全/护栏]

- [69] Saxena, Sharma, Kim, Dwivedula, Chen, Yang, Ravula, Hu, Akella, Angel, et al. On a foundation model for operating systems. arXiv:2312.07813, 2023. [系统经典/ML4SYS]
- [70] Shen, Zhong, Dwarkadas, Li, Stewart, Zhang. Hardware counter driven on-the-fly request signatures. ASPLOS XIII, pp. 189–200, 2008. [性能分析/遥测]
- [71] Snoek, Larochelle, Adams. Practical Bayesian optimization of machine learning algorithms. NeurIPS 2012 (Vol. 25). [优化方法经典]
- [72] Somashekar, Tandon, Kini, Chang, Husak, Bhagwan, Das, Gandhi, Natarajan. OPPerTune: Post-deployment configuration tuning of services made easy. NSDI '24, pp. 1101–1120. [先前调优器]
- [73] Su, Dhanotia, Torres, Gandhi, Gholkar, Kanaujia, Naumov, Subramanian, Andrei, Yuan, Tang. DCPerf: An open-source, battle-tested performance benchmark suite for datacenter workloads. ISCA '25, pp. 1717–1730. [基准/工具]
- [74] Tewari, Yadalam, Peters, Agarwal, Akella, Swift, Rossbach. OQueue: Observable communication in learning directed operating systems. 4th Workshop on Practical Adoption Challenges of ML for Systems, pp. 31–36, 2025. [安全/护栏]
- [75] Trummer. DB-BERT: a database tuning tool that "reads the manual". SIGMOD 2022, pp. 190–203. [LLM4SYS]
- [76] Van Aken, Pavlo, Gordon, Zhang. Automatic database management system tuning through large-scale machine learning. SIGMOD 2017, pp. 1009–1024. [先前调优器]
- [77] Vuppapalapati, Agarwal. Tiered memory management: Access latency is the key! SOSP '24, pp. 79–94. [系统经典/ML4SYS]
- [78] Wang, Li, Hoffmann, Lu, Sentosa, Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. ASPLOS '18, pp. 154–168. [先前调优器]
- [79] Wang, Li, Liang, Wu, Yan. Autothrottle: A practical bi-level approach to resource management for SLO-targeted microservices. NSDI '24, pp. 149–165. [先前调优器]
- [80] Watkins, Dayan. Q-learning. Machine Learning 8(3):279–292, 1992. [优化方法经典]
- [81] Xiang, Lin, Deng, Lu, Rao, Yuan, Wang. Nomad: Non-exclusive memory tiering via transactional page migration. OSDI '24, pp. 19–35. [系统经典/ML4SYS]
- [82] Yang, Saxena, Dwivedula, Mahajan, Chaudhuri, Akella. Canopy: Property-driven learning for congestion control. EuroSys 2026, pp. 1658–1673. [安全/护栏]
- [83] Zhang, Van Aken, Wang, Dai, Jiang, Lao, Sheng, Pavlo, Gordon. A demonstration of the OtterTune automatic database management system tuning service. PVLDB 11(12):1910–1913, 2018. [先前调优器]
- [84] Zhang, Lim, Butrovich, Pavlo. The Holon approach for simultaneously tuning multiple components in a self-driving database management system with machine learning via synthesized proto-actions. PVLDB 17(11):3373–3387, 2024. [先前调优器]
- [85] Zhang, Lim, Pavlo. This is going to sound crazy, but what if we used large language models to boost automatic database tuning algorithms by leveraging prior history? We will find better configurations more quickly than retraining from scratch! (Booster). arXiv:2510.17748, 2025. [LLM4SYS]
- [86] Zhang, Tune, Hagmann, Jnagal, Gokhale, Wilkes. CPI2: CPU performance isolation for shared compute clusters. EuroSys 2013, pp. 379–391. [性能分析/遥测]

[87] Zheng, Hu, Zhang, Quinn. Towards agentic OS: An LLM agent framework for Linux schedulers. arXiv:2509.01245, 2025. (SchedCP) [LLM4SYS]

[88] Zheng, Yang, Chen, Quinn. Kgent: Kernel extensions large language model agent. eBPF '24 (SIGCOMM Workshop), pp. 30–36. [安全/护栏]

[89] Zheng, Yu, Yang, Hu, Lai, Williams, Quinn. Extending applications safely and efficiently. OSDI '25, pp. 557–574. [安全/护栏]

[90] Zhu, Liu, Guo, Bao, Ma, Liu, Song, Yang. BestConfig: Tapping the performance potential of systems via automatic configuration tuning. SoCC '17, pp. 338–350. [先前调优器]

附录 C 仓库速览与复现指南

开源仓库：github.com/Columbia-DAP-Lab/SemaTune（曾用名遗留：README 项目名 SemaTune，CLI 入口 `os-param-tuning`，Python 包名 `barebones_optimizer`）。仓库为 Python 控制面，`src/` 约 10.6k 行，`docs/` 有 26 篇 MkDocs 文档，`config/examples/` 含 26 个示例配置。

C.1 目录结构与论文组件对照

路径	行数	论文组件
<code>dual_loop_optimizer.py</code>	306	双循环 Tuner（代码术语 Speculator/Actor）
<code>tuners/llm.py</code>	2,172	Context Manager（prompt 组装）+ LLM 前端 + replay + gist
<code>parameter_manager.py</code>	1,812	Parameter Validator：41 旋钮元数据 + 写参/快照
<code>optimizer.py</code>	1,426	单循环主控、校验应用、快照恢复、gist 落盘
<code>benchmark.py</code>	1,118	API-Telemetry：perf stat / RAPL / C-state / CPU load
<code>benchmarks/{sysbench,benchbase}.py</code>	1,072 / 638	应用指标适配器（sysbench、TPC-C/PostgreSQL）
<code>tuners/{mlos_tuner,bayesian,qlearning,dqn}.py</code>	合计 754	四个 baseline 调优器（薄封装外部库）
<code>tuners/llm_trimming.py</code>	395	TuxBot-Trim（LLM 压缩搜索空间）
<code>doctor.py</code>	100	主机前置自检 CLI

表 12-2 仓库关键文件与论文组件对照。

C.2 安装与自检

```
python3 -m venv .venv
source .venv/bin/activate
pip install -e ".[dev,docs,llm]"
# 装全部 baseline tuner（Bayesian/MLOS/DQN）：
# pip install -e ".[dev,docs,llm,tuners]"

.venv/bin/os-param-tuning doctor # 主机自检
.venv/bin/os-param-tuning --help
```

C.3 跑一个 session（最小路径）

LLM 后端需要自带 key（GEMINI_API_KEY 或 OPENROUTER_API_KEY，切勿写入 config 或提交）。无 key 时可先用 replay 模式走通全流程（llm_replay_file 按 iteration 重放提案，走完整校验与应用路径）：

```
sudo --preserve-env=PATH,OS_PARAM_TUNING_ROOT \
  OS_PARAM_TUNING_ROOT="$(pwd)" \
  .venv/bin/os-param-tuning run \
  --config config/examples/sysbench_cpu_fixed.json
# 查看结果（历史 JSON 在 results/ 下）：
jq '.history[] | {iteration, reward, parameters,
  throughput: .metrics.throughput}' results/<run>.json
```

【注意】

安全须知（README 原样精神）：SemaTune 会修改运行中的 Linux 内核与系统参数，请使用专用主机，运行前审查配置，sudo 时保留 OS_PARAM_TUNING_ROOT。恢复是 best-effort：kill -9、断电、主机崩溃不保证清理，docs/safety-and-restore.md 建议「正式实验前自行记录原值，重启往往是最干净的恢复路径」。

【零基础知识补给站】

第一次跑通的推荐路线（docs/quick-start.md 精神）：第一步，venv 安装后用 doctor 自检，它会检查 debugfs 挂载、cpufreq 驱动、sysctl 可写性等前置条件；第二步，跑 config/examples/sysbench_cpu_fixed.json（固定参数 tuner，不碰 LLM），验证负载挂接与遥测采集；第三步，用 replay 配置（config/replay/ 下有现成历史）验证 LLM 提案的解析、校验、应用全链路，不需要任何 API key；第四步，才接入真实 key 跑 config/examples/ 下的 llm 双循环示例。每步之间出了问题都知道怪谁——这是把复杂系统拆成「机制验证 → 数据通路验证 → 智能验证」三层递进的标准做法。跑 TPC-C 还需 BenchBase 依赖（本地副本 deps/benchbase/ 为空，需按 docs 指引另行获取构建）。

Insight#1 doctor + fixed + replay + live 的四步路径设计成熟，反映作者对「让陌生人 30 分钟跑起来」的认真。

Insight#2 41 旋钮元数据恰好 41 条（含 1 个 busy_poll 别名），与论文口径吻合；文档 docs/os-parameters.md 逐一列出写入路径与风险等级，是论文之外最有价值的参考。

Insight#3 跨 run 记忆（ChromaDB + embedding）未开源，复现 Figure 11 需自行实现或用 previous_run_gist 手动近似——见第 11 章对账。

附录 D 论文图表清单

论文全部 12 图 7 表的一句话索引，便于按图索骥。

图/表	页	内容一句话
Figure 1	p.2	稳态在线调优循环示意：host-side tuner 与 OS knobs 的信号往返，界定 out-of-band 研究对象
Table 1	p.3	MLOS 在 Memcached 500k QPS 下的踩坑轨迹：>47× 尾延迟、两次 minperfct > maxperfct 违例

图/表	页	内容一句话
Figure 2	p.4	左：IPC/Cache 代理目标使 p99 差 2×；右：调优面 1→32 参数 p99 恶化 50%（注：各柱具体值文本提取缺失，以引言口径为准）
Figure 3	p.5	TuxBot 系统总览：六步工作流（遥测→上下文→双循环→校验→应用）
Figure 4	p.6	Prompt 模板：稳定 session specification + per-iteration update
Figure 5	p.6	双循环时间线与 context entry 可见性/消费规则
Figure 6	p.10	主结果：11 子集稳定 +87.2% vs MLOS +50.5%；全集 +72.5% vs -31.9%
Figure 7	p.10	间接信号：仅系统指标 TuxBot +64.0%，超应用指标 MLOS 13.5pp
Figure 8	p.11	双循环成本–质量最优点：+87.2% @ \$0.20 vs 单 Reasoning +89.7% @ \$0.42
Figure 9	p.11	轨迹鲁棒性：P50 bad-window 率 16.9% vs 28.8%，variability 11.1% vs 25.1%
Table 2	p.11–12	事故率表：TuxBot 0.0%/0.0%，MLOS 16.7%/8.3%
Figure 10	p.12	维度扩展：41 旋钮 TuxBot +155.9% vs MLOS +13.0%
Table 3	p.12	决策延迟：MLOS 0.49s→5.73s（涨 10×），TuxBot Instant 保持 1.01–2.55s
Figure 11	p.13	记忆消融：Top-3 把稳定提升从 +144.7% 提到 +202.9%，Top-1 反而最差
Figure 12	p.13	后端互换：Gemini 2.5 +216.7% / Gemini 3 +152.7% / Kimi K2 +143.8% @ \$0.048
Table 4	p.19	附录 A：逐负载调优阶段提升分布（App/System/IPC 三组 Mean/SD/Range）
Table 5	p.19	附录 A：逐负载稳定阶段提升分布
Table 6	p.19	附录 B：13 负载 × 6 方法原始值与提升%（注：成对排版，文本提取有串位风险，引用请回查原表）
Table 7	p.18/20	附录 C：10 系统沿 Live loop/Indirect/Semantics-aware/Bounded 四属性定位（注：勾选矩阵文本提取丢失）

表 12-3 论文图表索引（两处不确定项已随文标注）。

【零基础知识补给站】

怎么使用这份清单做「证据审计」：拿到论文的任何结论性句子，先找到它引用的图/表，再问三个问题——这张图/表的数据口径是什么（全集还是 11 子集、调优期还是稳定期）？有没有已标注的不确定项（Figure 2 的柱值、Table 6 的串位、Table 7 的勾选矩阵都在本清单里标了注）？支撑它的消融在哪一节？TuxBot 论文的证据链相当完整：主结论（87.2%）有 Figure 6 + 附录 Table 6 双层支撑，安全结论（0 事故）有 Table 2 + Figure 9 双层支撑，泛化结论有 § 6.9 独立实验。三个标注了不确定项的图/表全部来自附录或动机章节，不在主证据链的关键路径上——这是审计后值得记录的整体判断。

Insight#1 12 图 7 表中仅 3 项存在提取不确定性，且都不在主证据链关键路径上，论文证据完整性评为良好。

Insight#2 Figure 6 / Table 2 / Figure 10 是全文的「三支柱」：性能、安全、可扩展，各自独立可验证。

Insight#3 复读者建议从 Table 2 入手重跑安全性实验——它对算力要求最低、结论最强、且最接近生产采纳者的真实关切。