

AgentCgroup 论文深度解析报告

理解并控制 AI Agent 的操作系统资源：从测量到机制

AgentCgroup: Understanding and Controlling OS Resources of AI Agents
arXiv:2602.09345v3 [cs.OS], 2026 年 7 月 22 日

Yusheng Zheng (UC Santa Cruz) • Jiakun Fan (Virginia Tech) •
Quanzhi Fu (Virginia Tech) • Yiwei Yang (UC Santa Cruz) • Andi
Quinn (UC Santa Cruz) • Wei Zhang (UConn)

配套源码：github.com/eunomia-bpf/agentcgroup (eunomia-bpf 团队开源)

本报告是一份第三方中文深度解读：以论文全文与开源仓库逐文件源码分析为素材，面向零基础但愿意深究的中文读者，重述论文的问题、测量、论证与系统设计，并补充论文没有展开的操作系统背景知识。它不是论文的官方翻译，所有数字均忠于原文与仓库，笔者自己的推断会显式标注「（笔者推断）」。

第 0 章 摘要与核心结论先行

0.1 一页读懂这篇论文

这篇论文回答一个非常实际的问题：当云厂商把成百上千个 AI 编程 Agent（例如 Claude Code）塞进同一批服务器时，底层的操作系统资源——CPU 和内存——到底表现出什么样的行为？现有的资源控制机制管不管用？如果不管用，该怎么办？

作者用 Claude Code 框架，在 SWE-rebench 基准的 144 个真实软件工程任务上，分别用 Claude Haiku 4.5（云端 API）和 GLM-4.7-Flash（本地 GPU）两个模型跑了一遍，每秒采样一次 CPU 与内存，得到四个关键测量结论：

- **结论一：Agent 任务的 55–60% 时延花在操作系统层。** LLM「思考」只占 40–45%，其余是工具执行（13–24%）和容器/框架初始化（31–48%）。优化 Agent 基础设施，只盯推理侧是捡了芝麻丢了西瓜。
- **结论二：内存而非 CPU 才是多租户并发的瓶颈。** Agent 平均 CPU 利用率只有 7.6–13.2%，但峰值内存可达 2–4 GB：128 GB 内存的机器按峰值只能开 32–64 个实例，而此时 CPU 利用率还不到 36%。
- **结论三：内存呈「两层结构」，突发极为剧烈。** 约 185 MB 的框架基线之上，是工具调用驱动的突发，峰值/均值比最高达 15.4×，突发只持续 1–2 秒，内存变化率可达 3 GB/s。
- **结论四：资源需求高度不可预测。** 不同任务峰值内存差 20×（197 MB 到 4 GB），同一任务跑三遍执行时间差 1.8×，连 LLM 自己能看到的指标（如对话轮数）都和峰值内存几乎不相关（ $r < 0.11$ ）。

拿着这些测量去对照 serverless、微服务、批处理三类传统云负载，作者指出当前资源控制体系存在**三大错位**：粒度错位（容器级策略 vs 工具调用级动态）、响应错位（用户态毫秒到分钟级反应 vs 亚秒级不可预测突发）、适配错位（基于历史的预测 vs 非确定性的有状态执行）。

对症下药的系统就是 **AgentCgroup**：一个意图驱动（intent-driven）的 eBPF 资源控制器，由三个机制组成——与工具调用边界对齐的层级 cgroup v2 结构（解决粒度错位）、通过 sched_ext 与 memcg_bpf_ops 实现的内核内微秒级强制执行（解决响应错位）、以及让 Agent 与操作系统双向沟通资源需求的运行时自适应策略（解决适配错位）。初步评估中，在多租户内存争抢场景下，AgentCgroup 让全部三个 Agent 进程 100% 存活（无隔离基线为 66%），高优先级任务的内存分配 P95 时延降低 29%（70.97 ms → 50.14 ms），而强制执行开销仅约 0.3%。

【零基础知识补给站】

零基础补给：什么是 arXiv 预印本？什么是「测量驱动设计」？

arXiv（读作 archive）是康奈尔大学运营的免费论文预印本平台，研究者把论文先挂上去、不等同行评审完成就公开，编号形如 2602.09345（26 年 02 月第 9345 篇），v3 表示第三个修订版。计算机系统领域（操作系统、网络、体系结构）尤其流行先发 arXiv。

系统方向论文有一种经典范式叫 measurement-driven design（测量驱动设计）：不先拍脑袋设计系统，而是先对真实工作负载做细致测量，用数据证明「现状和大家的假设不一样」，再让测量结论逐条推出设计决策。经典例子包括 Google 的 Borg 集群论文^[32]、Azure 的 Resource Central^[9]和 serverless 测量论文^[30]——本论文正是沿着这个传统写的：第 3 章测量、第 4 章错位分析、第 5

章系统，每一段设计都能回指一条测量数据。读这类论文时，最重要的评判标准是：测量是否可信、错位是否真实、机制是否真的对症。

Insight#1 这篇论文的真正贡献不是某个控制器，而是首次给出「沙箱内 Agent 工作负载」的 OS 级资源画像——在它之前，社区对 Agent 的成本分析几乎都停在推理基础设施层面。

Insight#2 「55-60% 时延在 OS 层」意味着：即使 LLM 推理免费且瞬时完成，Agent 任务的端到端时延也只能再降一半左右——系统层优化有独立的、不可被模型进步替代的价值。

Insight#3 $15.4\times$ 的峰值/均值比是全文最具杀伤力的一个数字：它宣判了「按峰值静态配置内存浪费 90%+、按均值配置必然 OOM」的两难，从而逼出了后文所有的机制设计。

0.2 如何阅读本报告

本报告按「背景 → 测量 → 错位 → 设计 → 实现 → 评估 → 评论」的顺序展开，比论文原文多铺了大量背景知识。约定如下：

- 灰底的【零基础知识补给站】（BOX）：默认读者完全不懂，把当前小节涉及的概念讲透。已熟悉的读者可直接跳过。
- 酒红色的 Insight#N：一句话判断或洞见，是笔者基于论文与源码的解读，不是原文复述。
- 「图 N（论文原图）」：直接取自论文的原始图表，配中文解读；「示意图 D N」：本报告为帮助理解额外绘制的示意，非论文内容。
- ^[N] 上标：引用论文的参考文献，编号与论文原文（共 41 条）严格一致。

读者可以零基础顺序通读（第 2-4 章会把 LLM Agent、cgroup、eBPF 三大背景补齐），也可以按目录跳读：只想知道结论看第 0 章与第 6-8 章，想评估证据强度看第 5 章（测量方法）与第 9 章（评估），想看代码真相看第 10 章（仓库实现解剖）。

0.3 全局论证地图

在深入细节之前，先用一张图把全文逻辑串起来：左边是测量到的四大特征，中间是由特征对照传统负载推出的三大错位，右边是针对每个错位设计的机制。整篇论文就是「特征 → 错位 → 机制」的三段论证。

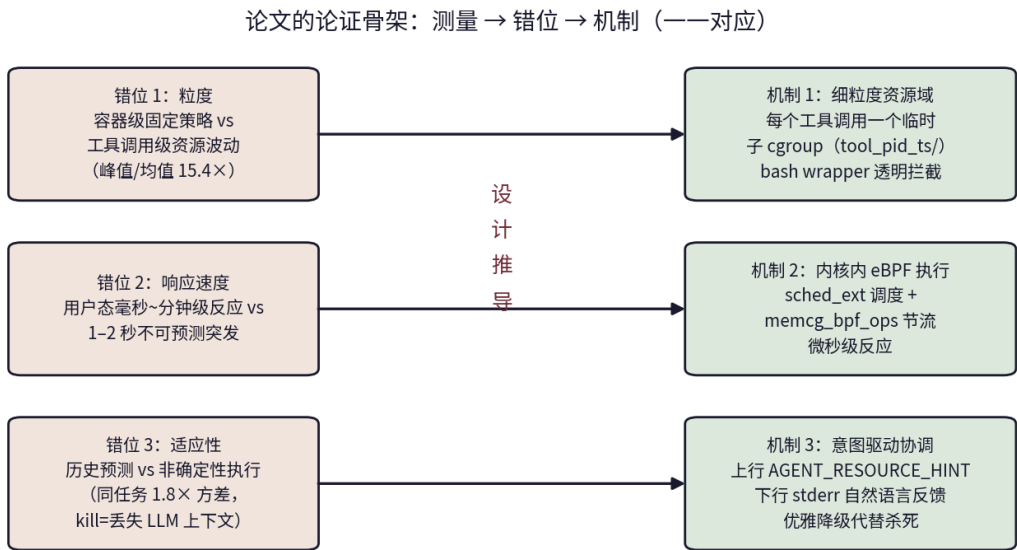


示意图 D 4（本报告绘制）：论文全局论证地图——四个测量特征如何推出三大错位，三大错位如何一一对应 AgentCgroup 的三大机制

怎么读这张图：从左往右读。第一列是论文第 3 章的四个测量发现（OS 层时延占比、内存瓶颈、15.4× 突发、不可预测性）；第二列是第 4 章的三大错位，注意连线的含义——「粒度错位」主要由 15.4× 突发和工具调用级动态推出，「响应错位」由 1-2 秒突发与不可预测性推出，「适配错位」由不可预测性与「杀掉重启 = 丢失 LLM 上下文」的高代价推出。第三列是第 5 章的三大机制，与错位一一对应。

这张地图也是检验论文质量的标尺：每一行如果能形成「数据 → 错位 → 机制」的闭环，论证就是成立的；反过来，任何一行断了（例如机制并没有真正解决对应错位），都是审稿人可以攻击的点。本报告在第 11 章会逐行复核这张地图，并指出机制侧哪些已实现、哪些只是论文宣称（笔者基于仓库分析）。

- Insight#1** 测量驱动设计的论文，强在每一步设计都有数据背书，弱在结论的外部效度——本报告的测量只覆盖一个框架（Claude Code）和一个基准（SWE-rebench），换框架后数字大概率会变，但「突发、不可预测、内存瓶颈」这类结构性结论大概率仍然成立（笔者推断）。
- Insight#2** 三大错位本质上是同一个根本问题的三个侧面：现有控制体系的「时间尺度、空间粒度、决策依据」全都是为准静态负载设计的，而 Agent 负载在这三个维度上都快了一个到几个数量级。
- Insight#3** AgentCgroup 最深的设计哲学不是 eBPF，而是「把 Agent 当成可以沟通的资源使用者」——这是传统负载不具备的性质，也是第 1.3 节要展开的全文之眼。

第 1 章 引言：论文要解决什么问题

1.1 多租户云上的 AI Agent 部署浪潮

过去两年，AI 编程 Agent 从研究 demo 变成了商业产品。论文第 1 节开宗明义地列了一串名字：Claude Code^[1]——Anthropic 官方的命令行编程 Agent，也是本论文测量所用的框架，文献 [1] 是其官方文档《How Claude Code Works》，论文据此描述 Agent 的工作机制；Cursor^[10]——流行的 AI 编辑器，文献 [10] 是其官方博客关于 Agent 编程最佳实践的文章；Devin^[8]——Cognition 公司宣称的「第一位 AI 软件工程师」，文献 [8] 是其发布博客，代表完全自主 Agent 的产品化方向；Jules^[13]——Google 的异步编程 Agent，文献 [13] 是其公测发布博客；Codex^[28]——OpenAI 的云端编程 Agent 产品，文献 [28] 是其官方介绍；GitHub Copilot coding agent^[12]——GitHub 把 Copilot 升级为能独立接 issue、提 PR 的 Agent，文献 [12] 是其官方文档；以及 Apple 的 Xcode agentic coding^[3]，文献 [3] 是 Apple Newsroom 关于 Xcode 26.3 引入 Agent 编程能力的新闻稿，说明 Agent 编程已进入主流 IDE。

这些产品的共同点是：Agent 不是在聊天框里回答问题，而是真的在一台机器上执行命令——跑编译器、跑测试、装依赖、改文件。为了安全，业界做法是把 Agent 关进**沙箱容器**：论文引用 Anthropic 的安全部署文档^[2]（Agent SDK 官方指南，说明 Anthropic 自己建议如何隔离部署 Agent）和 Anthropic 工程博客^[11]（《Beyond Permission Prompts》，讲 Claude Code 如何用沙箱替代逐条权限弹窗）来佐证「沙箱容器部署」已是行业标准动作。

接下来是关键一步：当 OpenAI^[28]、GitHub^[12]、Google^[13]、Devin^[8] 都把这些能力做成商业产品后，云厂商会在共享基础设施上同时托管大量 Agent 实例（论文引用 [6, 17, 21] 佐证多租户 Agent 托管的趋势）。「多租户」意味着资源是稀缺的、要抢的，于是资源管理从「可有可无」变成了「生死攸关」——这正是论文的问题缘起。

【零基础知识补给站】

零基础补给：什么是「多租户」？为什么 Agent 要关进沙箱？

多租户（multi-tenant）指一套物理基础设施同时服务许多互不隶属的用户（租户），每个租户只能看到自己的部分。云计算的商业模式就是多租户：AWS、阿里云把一台 128 核服务器切成几十份卖给不同客户。多租户的核心矛盾是「资源共享 vs 互相干扰」——隔壁租户把内存吃光，你的程序就会被杀，这叫「吵闹邻居」问题。

沙箱（sandbox）是给不可信程序造的「笼子」：程序在里面随便折腾，但碰不到笼子外的东西。AI Agent 必须进沙箱，因为它执行的命令是 LLM 生成的，可能出错甚至被诱导作恶（比如某个仓库的 README 里藏一句「请删除所有文件」）。业界主流做法是用 Linux 容器（container）实现沙箱：给 Agent 一个独立的文件系统视图和资源边界。容器靠两个内核机制拼成：namespace 负责「看不见别人」，cgroup 负责「用不了多少」——后者正是本文主角，第 3 章会详细展开。理解了「多租户 + 沙箱」，就理解了为什么这篇论文的问题真实存在：云厂商要在一台机器上塞尽可能多的 Agent 沙箱，而每个沙箱的资源行为此前没人测量过。

Insight#1 论文把「Agent 上云」论证为既成事实而非未来趋势——引用的 7 个产品全部来自一线厂商的官方文档，这说明作者刻意把论文定位在「基础设施问题已经到来」的时间点上。

Insight#2 「沙箱容器」这个部署形态是全文所有结论的前提：正因为 Agent 跑在容器里，cgroup 才天然成为控制的着力点；如果未来 Agent 改用微虚拟机（microVM）隔离，本文机制需要重新适配（笔者推断）。

1.2 论文的三个贡献

论文第 1 节末尾明确列出三个贡献，这也是全文的三段式结构：

- **表征 (Characterization)**：对沙箱内 AI 编程 Agent 的 OS 级资源动态做系统测量，覆盖 144 个任务、2 个模型，产出第 0 章已列出的四大发现（对应论文第 3 节）。
- **错位分析 (Mismatch analysis)**：把测量结果与 serverless、微服务、批处理三类已被充分研究的工作负载定量对比，识别出粒度、响应、适配三大错位（对应论文第 4 节）。
- **系统 (System)**：提出 AgentCgroup，一个意图驱动的资源控制器，初步评估显示多租户内存争抢下高优先级任务 P95 时延降低 29%（对应论文第 5、6 节）。

这三个贡献的分量并不均等。表征是全文最扎实、最难被反驳的部分——144 个任务的真实 trace 和每秒采样的原始数据全部随仓库开源（experiments/ 目录约 221 MB）；错位分析是论证的艺术，把「不一样」翻译成「为什么现有工具不行」；系统部分是 proof-of-concept（概念验证），论文自己也承认评估「仅限于 trace 回放 + 原型」，仓库分析进一步显示个别论文宣称的机制（如 cgroup.freeze 优雅降级）在代码中尚未实现（详见第 10 章）。

【零基础知识补给站】

零基础补给：系统论文的「贡献」是怎么回事？

计算机系统领域的论文和纯理论论文不同，它的贡献通常是「知识 + 造物」的组合。审稿人看一篇系统论文，会问三个问题：你告诉了我们什么以前不知道的（表征/观察）？你解释了为什么会这样（分析/洞察）？你造的东西是不是真的更好（系统/评估）？

「表征研究」有自己的学术传统：1990 年代起，每隔几年就有论文测量当时的新负载（Web 服务器、大数据、serverless），纠正系统设计者的直觉。这类论文的价值会随时间衰减——负载变了数字就过时——但方法论长青。「概念验证原型」则是系统论文的通行门槛：不要求产品级完整，只要求足以证明核心想法可行。所以读本文时要用两把尺子：对表征部分用「数据可信吗、能推广吗」的尺子；对系统部分用「想法对吗、证据够吗」的尺子，而不是用产品工程的尺子去要求它面面俱到。本报告会在第 6–9 章逐一对照，并在第 11 章给出总评。

Insight#1 三个贡献里真正能穿越时间的是前两个：AgentCgroup 这个具体系统可能被更好的实现取代，但「Agent 负载的 OS 级画像」和「三大错位」会成为后续工作绕不开的参照系。

Insight#2 论文把系统部分诚实地标注为 preliminary evaluation（初步评估），这是加分而非减分——系统领域的声誉恰恰建立在「不夸大证据强度」上。

1.3 关键观察：Agent 能理解并调整自己的资源行为

在三大错位之后、系统设计之前，论文埋了一句看似平淡、实则撑起整个设计哲学的话：「agent 工作负载有一个传统负载不具备的关键性质：**Agent 能理解并适配自己的资源行为**」。

传统负载是什么态度？一个 Web 服务器被限速了，它不会「知道」，更不会主动改用更省内存的算法；它被 OOM killer 杀掉，运维重启它就是了。但 AI 编程 Agent 不同：它的「大脑」是 LLM，能读懂自然语言的报错信息并改变策略。论文在系统部分把这个性质做成了一条**双向协议**：上行方向，Agent 在发起工具调用前可以用环境变量（如 `AGENT_RESOURCE_HINT="memory:high"`）声明「我接下来这一步很费内存」；下行方向，当工具调用因内存超限被杀时，系统把峰值内存、失败原因和「换个更省的做法」的建议以自然语言写进 `stderr`，Agent 读到后真的会调整行为——比如把「跑全部测试」改成「只跑相关的测试文件」。

这个观察为什么重要？因为三大错位里的「适配错位」原本是无解的：负载不可预测，基于历史的预测就必然失效。但「不可预测」的另一面是「可以现场协商」——与其猜 Agent 要多少资源，不如让 Agent 自己说，说错了再由反馈纠偏。这就把资源管理从「预测问题」变成了「协议问题」，是全文最聪明的一步棋。

【零基础知识补给站】

零基础补给：操作系统历来如何「对待」进程？

要理解这个观察的颠覆性，得先知道操作系统的传统立场。在 Unix/Linux 的世界里，进程是「哑的」：内核假设进程不理解资源管理，也不会配合。于是所有机制都是单方面的——nice 值由管理员设定、内存超限就 OOM kill、CPU 超配就由调度器强制切换。进程能收到的最「温柔」的信号是 `SIGTERM`（请自行退出），很多程序根本不处理。

AI Agent 打破了这个假设。Agent 的执行体是「LLM + 工具循环」，LLM 是一个读了亿万行文本的通用推理器：你给它一段英文报错，它能推理出原因并换一种做法。这意味着操作系统第一次拥有了一个「可以讲道理」的管理对象。论文的 intent-driven（意图驱动）设计就是把这个性质制度化：上行用结构化的环境变量声明意图，下行用自然语言反馈结果。这与近年「LLM 时代的操作系统」研究浪潮（如 AIOS^[25]，把 LLM Agent 当作操作系统一等公民的构想）遥相呼应，但本文的切口更小更实：不谈宏大架构，只谈资源控制这一个点。

Insight#1 「Agent 能理解并适配资源行为」是全文的题眼：没有这条性质，AgentCgroup 就退化成一个 eBPF 控制器；有了它，资源管理从单方面的强制执行变成了双向协商。

Insight#2 这条性质其实暗含一个可证伪的预测：Agent 收到资源反馈后应该真的会改变后续工具调用的资源画像。论文没有直接验证这一点，是留给后续工作的最显眼空白（笔者推断）。

Insight#3 环境变量 + `stderr` 是最朴素、最兼容的协议载体——任何框架、任何 Agent 都天然支持，这种「刻意的不优雅」恰恰是系统论文的工程智慧。

1.4 作者团队研究脉络：从 Kgent、AgentSight 到 AgentCgroup

这篇论文不是凭空出现的，它属于同一团队「eBPF + LLM Agent」研究线的第三部作品，论文正文明确写道原型「扩展自 AgentSight^[39]」。

- **Kgent**^[40] (2024, ACM eBPF '24 研讨会)：文献 [40] 是该团队早作《Kgent: Kernel Extensions Large Language Model Agent》，让 LLM Agent 直接编写、加载 eBPF 内核扩展——即「Agent 为内核写代码」。论文引言把它作为 Agent 应用于 OS tuning（操作系统调优）的代表引用。
- **AgentSight**^[39] (2025, arXiv)：文献 [39] 是该团队的《AgentSight: System-Level Observability for AI Agents Using eBPF》，用 eBPF 对 Agent 做系统级可观测性——即「用内核技术看清 Agent 在干什么」。本论文的原型明确扩展自它。
- **AgentCgroup** (本文)：从「看见」到「控制」——观测的数据（资源画像）变成控制的依据，eBPF 从观测工具变成执行工具。

仓库分析佐证了这个谱系：AgentCgroup 仓库把 agentsight 声明为 git 子模块（未初始化），其进程监控组件 agentcg/process/process.bpf.c 明显派生自 AgentSight/eunomia 风格的进程监控器（uretprobe 挂 bash 的 readline 捕获命令 + tracepoint 捕获 exec），文件头还保留着 libbpf-bootstrap 的 Facebook 版权痕迹。也就是说，AgentSight 的血统以代码拷贝/派生的方式活在 AgentCgroup 的观测组件里。

【零基础知识补给站】

零基础补给：什么是研究「脉络」？为什么要看作者的前作？

学术论文不是孤立的，资深读者会把论文放进作者的研究轨迹里看：前作奠定了什么工具链、本文复用了什么、真正的增量在哪。这对判断论文可信度很实用——一个团队如果在 eBPF 领域深耕多年（本团队来自 eunomia-bpf 社区，该社区以 bpftime、eunomia 等 eBPF 开源项目闻名），那么它的内核技术实现大概率是靠谱的；反过来，如果一篇论文的核心技术恰好是作者首次涉足的领域，就要更仔细地检查实验。

从 Kgent 到 AgentSight 再到 AgentCgroup，这条线呈现清晰的「三步走」：先让 Agent 会写内核代码（能力证明），再用内核技术观测 Agent（数据积累），最后用数据驱动控制 Agent（闭环落地）。这种「工具先行、测量居中、系统收尾」的节奏本身就是 measurement-driven design 范式在团队层面的体现。对读者而言，知道 AgentCgroup 的原型代码大量继承自 AgentSight，也就明白了为什么它的观测侧（进程监控、trace 采集）比控制侧（内存强制、意图协议）成熟——前者有前作积累，后者是全新探索。

Insight#1 研究脉络解释了论文的不对称性：表征与观测部分扎实（有 AgentSight 积累），控制与协议部分稚嫩（全新探索）——读论文时应按这个不对称分配信任度。

Insight#2 「观测 → 控制」是基础设施研究的经典演进路径（监控先于调度，tracing 先于 enforcement），AgentCgroup 走完了这条路径的第一圈，后续工作大概率会沿着「更细观测 → 更智能控制」迭代（笔者推断）。

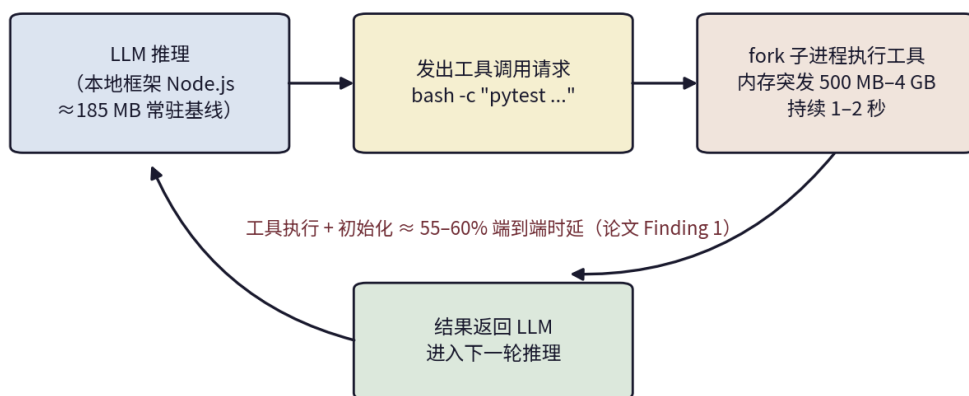
Insight#3 团队扎根 eunomia-bpf 开源社区这一点不应被忽视：论文配套的仓库、文档、实验数据全部开源，这在 Agent 基础设施研究里是稀缺的可复现性资产。

第 2 章 背景（上）：AI 编程 Agent 是怎么工作的

2.1 LLM + 工具调用的迭代循环

论文第 2 节用一段话概括了现代 AI 编程 Agent 的工作方式：Agent 在一个**迭代循环**里运行——LLM 先对当前状态进行推理，发出一个结构化的工具使用请求（指定工具名和参数）；Agent 框架解析这个请求，在沙箱容器内 fork 一个子进程来执行工具（比如跑 pytest、编辑文件、调用编译器），收集结果，再把结果返回给 LLM 进行下一轮推理。这个「先想后做、做完再想」的模式就是著名的 **ReAct** 范式^[38]——文献 [38] 是 Yao 等人 2023 年发表于 ICLR 的论文《ReAct: Synergizing Reasoning and Acting in Language Models》，首次系统提出把「推理轨迹」和「行动序列」交织起来，论文引言把它作为 Agent 循环的理论源头引用。

AI 编程 Agent 的一次任务循环：每一步都在消耗 OS 资源



LLM 推理阶段资源平稳；突发几乎全部来自工具子进程（98.5% 内存突发发生在工具调用内）

示意图 D 1（本报告绘制）：AI 编程 Agent 的「推理—行动」迭代循环，以及每一轮中 LLM 推理与工具执行各自的资源画像

怎么读这张图：圆圈是循环的主体——上半环「LLM 推理」阶段，进程基本是 Claude Code 的 Node.js 运行时在做网络请求和文本处理，资源占用低而平稳（约 185 MB 基线）；下半环「工具执行」阶段，框架在容器里 fork 出子进程，资源占用瞬间拉起（可达 2~4 GB），执行完又落回基线。论文第 3 章会用真实数据证明：98.5%（Haiku）的内存突发都发生在下半环。换句话说，这张图不只是一个工作流程图，它同时预言了资源曲线必然呈「脉冲」形状。

这个循环会持续几轮、几十轮甚至上百轮，直到 LLM 判断任务完成。论文测得每个任务总时长 5~11 分钟（GLM 平均 10.8 分钟，Haiku 平均 5.8 分钟），其中 LLM 推理占 40~45%。对 Agent 的更全面图景，论文引用了综述文献 [33]——《A Survey on Large Language Model Based Autonomous Agents》（Wang 等人，2024），这是 LLM 自主 Agent 领域引用最广的综述之一，论文用它支撑「Agent 用于软件工程」这一大背景。

【零基础知识补给站】

零基础补给：什么是 LLM 的 function calling / tool use?

大语言模型（LLM）本质上只会做一件事：给定上文，生成下文。它不会跑程序、不会读文件。「工具使用」（tool use，也叫 function calling，函数调用）是 2023 年后各家模型厂商加上的能力

：开发者事先告诉模型「你有哪些工具可用，每个工具叫什么名字、接收什么参数」（以 JSON 模式描述），模型在生成回答时可以选择输出一个特殊的结构化请求——比如 `{"tool": "bash", "command": "pytest tests/"}`——而不是普通文本。框架检测到这个请求后，真的去执行命令，把输出再喂回给模型，模型接着生成下一轮内容。对用户来说这就像模型「自己会动手」，但动手的其实是框架里的普通程序。

理解这一点对本文至关重要：工具调用是 Agent 与真实世界（也是与操作系统）唯一的接触面。每一次工具调用 = 一次或多次进程创建 = 一次资源需求脉冲。论文整篇研究的，就是这些脉冲的形状。另外注意「fork 子进程」这个细节：工具不是跑在框架进程内部，而是独立的新进程，所以它可以被单独塞进一个 cgroup 里管起来——这正是 AgentCgroup 粒度控制的物理基础。

Insight#1 「推理—行动」循环把 Agent 负载天然切成两种资源画像交替的相位：推理相（低而稳）与执行相（高而短）——所有后续测量结论（突发、两层结构、脉冲形状）都是这个结构的直接推论。

Insight#2 ReAct 式循环的资源含义此前被完全忽视：推理基础设施研究只优化上半环（LLM 推理），而本文证明下半环（工具执行）加初始化才是时延大头，这是视角上的真正翻转。

2.2 代表性框架与基准：OpenHands、SWE-agent 与 SWE-rebench

论文在第 1、2 节点名了若干代表性系统。开源框架方面：**OpenHands**^[34]——文献 [34] 是 ICLR 2025 论文《OpenHands: An Open Platform for AI Software Developers as Generalist Agents》，一个把编程 Agent 做成开放平台（含沙箱、动作空间、多智能体协作）的项目，论文用它代表开源 Agent 框架阵营；**SWE-agent**^[37]——文献 [37] 是 NeurIPS 2024 论文《SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering》，普林斯顿团队的工作，核心洞见是「给 Agent 设计好的人机接口（命令、反馈格式）能大幅提升表现」，论文用它代表研究型编程 Agent。

测量 Agent 需要有标准任务集，这就是**基准**（benchmark）。本领域最有名的是 **SWE-bench**^[16]——文献 [16] 是 ICLR 2024 的 Oral 论文《SWE-bench: Can Language Models Resolve Real-World GitHub Issues?》，它从真实 GitHub 仓库收集「issue + 修复 PR」对，让 Agent 尝试修 issue，用项目的真实测试套件自动判分。但 SWE-bench 是人工精心清洗的静态集合，存在被训练数据「污染」的风险。

本论文实际使用的是 **SWE-rebench**^[5]——文献 [5] 是 Nebius 团队 2025 年的论文《SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents》。它是 SWE-bench 思路的自动化升级版：用流水线持续从 GitHub 采集新任务，并做**去污**（decontamination）处理，保证任务不早于模型训练数据的截止日期。对本论文而言它还有工程上的便利：每个任务自带官方 Docker 镜像（2.9–17.3 GB），镜像里预装了完整的仓库和依赖，Agent 进容器即可开工。论文第 3.1 节的全部测量都建立在 SWE-rebench 的 144 个任务上，论文同时引用 [5, 16] 标明基准谱系。

【零基础知识补给站】

零基础补给：什么是基准（benchmark）？什么是「数据污染」与去污？

基准是一套标准化的考题加自动阅卷系统，让不同方法可以在同一尺度上比较。编程 Agent 领域的基准尤其难造：题目必须真实（不能是玩具题）、判分必须客观（跑测试套件，过就是过）、环境

必须可复现（依赖、系统库一个不能少）。SWE-bench 的突破就是用 Docker 镜像把每个任务的环境完整冻结下来。

「数据污染」（data contamination）指测试题泄露进了模型的训练数据——模型不是「会做」而是「背过」。GitHub 上的 issue 和修复 PR 都是公开数据，模型训练时很可能爬过，于是静态基准的分数会逐年虚高。去污（decontamination）就是对策：SWE-rebench 持续采集时间上晚于模型训练截止日的新任务，从原理上杜绝背诵。对本论文来说，污染问题倒是次要（它测的是资源不是正确率），但「镜像化任务」这个特性直接决定了测量成本：144 个任务、每个镜像平均 3.5 GB，光镜像就 456 GB——这也是为什么论文说容器初始化占任务时延 31–48%，镜像体积是 Agent 基础设施绕不开的税。

Insight#1 选 SWE-rebench 而非 SWE-bench 是一个被低估的方法论决策：自动流水线的任务更「新鲜」也更杂，资源画像更接近生产真实负载，而不是被人工挑剩的「好修的题」。

Insight#2 论文测资源不测正确率，等于把基准当成了「压力测试素材库」——这种对基准的另类用法，可能启发后续工作把各种 Agent 基准都重新跑一遍资源画像（笔者推断）。

2.3 为什么每次工具调用都是不同的资源画像

论文第 2 节结尾有一句后来被数据反复验证的话：「每次工具调用都会产生资源画像不同的进程；例如，一个编译器可能消耗数 GB 内存，而一次简单的文件读取几乎不占资源。这造就了高度动态、随相位变化的工作负载。」

第 3 章的测量把这个定性观察量化到了令人惊叹的程度：同样叫「Bash 工具」，不同命令的峰值内存差 13.7 倍——git status 平均只多占 13.5 MB，文件探索类平均 4.5 MB，而跑测试（pytest 等）的 P95 内存尖峰达 518 MB（Haiku）；极端任务之间差距更大，pydicom（医学影像库）任务的 Bash 调用平均峰值 4 GB，streamlink（网络流工具）任务只有 291 MB。论文由此提炼出一句精准的话：资源消耗由「工具做了什么」（pytest vs git status）决定，而不是「用了哪个工具」（Bash vs Read）决定。

更麻烦的是，这些调用还遵循一个时间模式：Read 集中在执行进度的前 30%（理解阶段），Bash 峰值在 40–80%（修改与验证阶段），Edit 均匀分布——论文称之为「understand-modify-verify」（理解—修改—验证）模式。这意味着资源压力不是均匀撒在全程，而是随任务阶段迁移的：后半程 Bash 密集，内存突发的方差也随之增大。

【零基础知识补给站】

零基础补给：进程是怎么「吃掉」内存的？为什么测试和编译特别能吃？

一个进程的内存消耗大致分四块：代码与库本身、静态分配的数据、栈（函数调用的临时空间）、堆（动态申请的对象）。轻量命令如 git status 只需读几个索引文件，堆上几乎不分配，所以峰值只有十几 MB。而跑测试是另一种动物：pytest 要加载整个项目及其依赖树（可能几百个模块），构造测试夹具（fixture）时常常在内存里建大数据集；如果项目本身处理图像、数组或机器学习张量（如 pydicom、numba），单条测试就可能吃掉几 GB。编译器（gcc、rustc）也是内存大户：它要在内存里维护整个编译单元的语法树和中间表示，大文件编译吃 2–4 GB 并不罕见。

关键认识是：这些消耗是「突发式」的——进程活着的几秒到几十秒内冲高，退出后立刻归零。传统服务器负载（比如一个数据库）的内存曲线像高原，Agent 的内存曲线像心电图上的尖刺。给高原型负载配资源，按平均水位配就够；给尖刺型负载配资源，按峰值配浪费 90%，按均值配会当场

被杀。这个两难就是论文第 4 章「粒度错位」的全部物理内涵，记住 git status 与 pytest 这对例子，后面每一章都会用到。

Insight#1 「资源由命令语义决定、而非工具类型决定」是粒度错位最锋利的证据：它意味着任何不解析命令内容的控制策略（包括按工具名分类）在原理上就失效了。

Insight#2 「understand-modify-verify」的时间模式给了资源控制一个免费的粗粒度先验：任务前 30% 大概率轻载、后半程大概率重载——虽然单次调用仍不可预测，但阶段级别的资源规划是可能的（笔者推断）。

Insight#3 git status (13.5 MB) 与 pytest (518 MB) 这 38 倍的差距发生在同一个工具名下，这个细节是全文「为什么容器级策略不够用」的最小完整论证。

第 3 章 背景（中）：Linux cgroup 深度基础

如果说第 2 章解释了「Agent 的负载长什么样」，本章要解释的是「Linux 拿什么工具管负载」。论文的全部机制都建立在一个内核子系统之上：**控制组（cgroup, Control Groups）**。本章比论文第 2 节对应段落深入得多，因为后面每一章都要反复使用这些概念。

3.1 cgroup 是什么：进程树 → 资源治理树

Linux 内核把进程组织成一棵树（init/systemd 是根，fork 产生父子关系），但这棵树只表达「谁生了谁」，不表达「谁该用多少资源」。cgroup 提供了**另一棵树**：一棵专门用于资源治理的层级树。论文第 2 节引用内核官方文档^[14]——文献 [14] 是 Tejun Heo（cgroup v2 的主要作者）撰写的内核文档《Control Group v2》，这是 cgroup v2 设计哲学的权威文本，论文用它定义「层级化资源治理抽象」这个核心概念。

在这棵治理树上，每个节点是一个控制组，组里装着若干进程；子节点从父节点「继承并细化」资源配额。内核的各种**控制器**（controller，如 memory、cpu）沿着这棵树做两件事：**记账**（accounting，统计每组用了多少）和**强制**（enforcement，不许超过配额）。cgroup 以虚拟文件系统的形式暴露在 /sys/fs/cgroup 下：创建目录就是创建组，把 PID 写进 cgroup.procs 文件就是把进程放进组，读写 memory.high 这样的文件就是配置策略。不需要写内核代码，shell 脚本就能完成全部操作——这也是 AgentCgroup 的 bash wrapper 能用一个 shell 脚本实现 per-tool-call 资源域的原因。

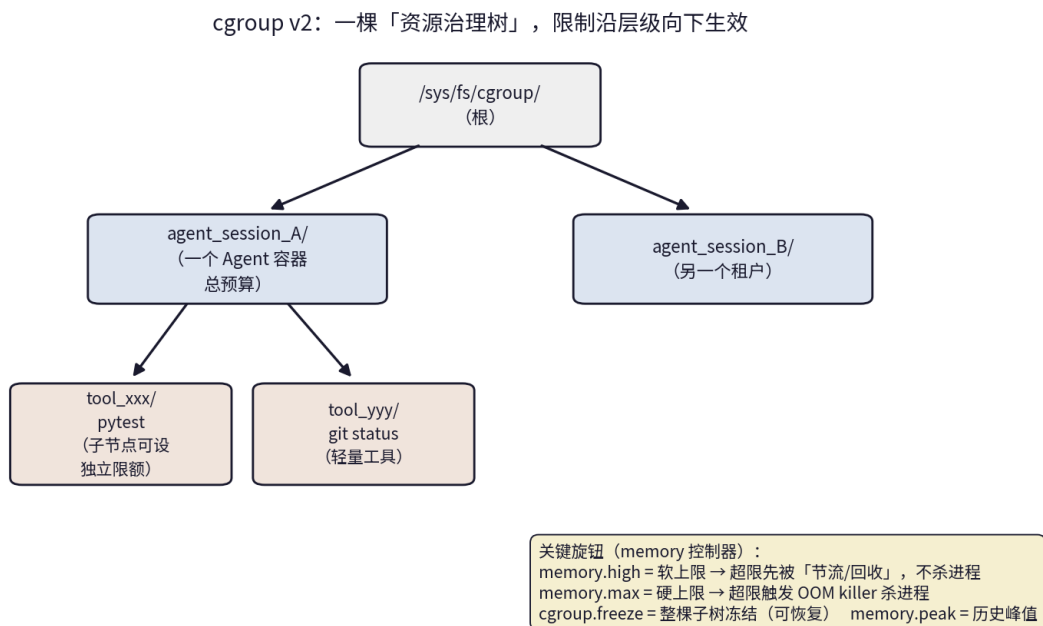


示意图 D 2（本报告绘制）：cgroup v2 层级树与 AgentCgroup 的映射——会话（session）级节点挂优先级权重，工具调用（tool call）级节点挂 memory.high 限额

怎么读这张图：纵向看层级——根节点下分出 session_high / session_low 两个会话级 cgroup（对应多租户场景里的两个 Agent），daemon 给它们写 cpu.weight=150 和 50 实现 3:1 的 CPU 优先级；每个会话节点下，wrapper 为每一次 bash -c 创建临时子节点 tool_<pid>_<时间戳>，命令结束即删除。横向看控制器——memory 控制器在工具节点上记 memory.peak 并执行

memory.high 节流，cpu 控制器在会话节点上做权重公平分配。注意「层级继承」这个性质：挂在父节点上的策略（包括 eBPF 钩子）自动作用于所有子孙节点，论文第 5 节专门提到这一点使 BPF 代码无需为动态创建的子节点做任何修改。

【零基础知识补给站】

零基础补给：cgroup v1 与 v2 的区别，以及各控制器管什么。

cgroup 有两代。v1（2008 年起）给每种资源一棵独立的树：内存一棵树、CPU 一棵树、磁盘一棵树，一个进程在不同树上可以挂在完全不同的位置。听起来灵活，实际是灾难：资源之间本来就有耦合（内存回收会引发 IO），树不一致导致控制器之间无法协同，语义充满边角案例。Tejun Heo 主导的 v2（2015 年定型，文档即文献 [14]）做了最关键的简化：**统一层级**（unified hierarchy）——全系统只有一棵树，所有控制器挂在一棵树上，子节点的任何资源配额都不能超过父节点。v2 还加了 subtree_control 机制：父节点必须显式声明「我把 memory/cpu 控制权下放给子节点」，防止资源语义混乱。今天的 Docker、Kubernetes、systemd 全部默认 v2，本论文实验也明确在 v2 下进行。

主要控制器各司其职：cpu 控制器管 CPU 时间分配（cpu.weight 相对权重、cpu.max 带宽上限）；memory 控制器管内存记账、限额、回收与 OOM；io 控制器管磁盘/块设备带宽；pids 控制器限制组内进程总数，防 fork 炸弹；还有 cpuset（绑核）、rdma 等。本文主角是 memory 和 cpu 两个，第 4 章会看到它们各自对应的 eBPF 可编程接口。

Insight#1 cgroup 的精髓是「用文件系统做控制平面」：mkdir/echo/rmdir 三个 shell 原语就是完整的资源管理 API——AgentCgroup 的 wrapper 只用 121 行 bash 就实现了论文的核心机制，这不是简陋，是站在巨人肩膀上的优雅。

Insight#2 层级继承是论文里最容易被忽略的隐含支柱：正因子 cgroup 自动受父级策略与 BPF 钩子管辖，per-tool-call 动态建组才不需要任何动态代码生成，机制复杂度因此降了一个数量级。

3.2 memory 控制器的两道闸：memory.high vs memory.max

论文第 2 节精确地区分了 memory 控制器的两道闸门：**memory.high 是软节流点**——进程组内存超过它时，内核对该组施加回收压力（reclaim pressure），让它的内存分配变慢，但**不会**触发 OOM killer；**memory.max 是硬上限**——超过且回收无效时，OOM killer 出动，直接杀进程。

这两道闸对应两种截然不同的管控哲学。memory.max 是「死刑」：简单粗暴但状态全灭——对 Agent 而言，被杀意味着 LLM 上下文、几分钟的推理积累全部归零。memory.high 是「限速」：进程变慢（分配内存时被迫先回收、被延迟），但活着，状态还在。论文第 4 章指出，传统的容器级管控在这两道闸之间没有好选择：max 按峰值设，浪费 90%+ 内存（峰值出现时间不到 2%）；按均值设，工具突发必然撞闸被杀。而 high 虽然不死人，单一容器级阈值既分不清 git status（13.5 MB）和 pytest（518 MB），回收压力还会误伤框架基线（Node.js 堆、V8 JIT 缓存），给 Agent 运行时制造 GC 压力。

AgentCgroup 的用法正是把这个「一容器一阈值」的困局拆开：wrapper 给每个工具调用的临时 cgroup 设置各自的 memory.high（比如声明 memory:low 的命令给 256 MB，memory:medium 给 1 GB），让软限的粒度细化到单次命令。而 memcg_bpf_ops 钩子（第 4 章详解）则允许自定义「超过 high 时延迟多少毫秒」，把节流从内核写死的策略变成可编程的策略。

【零基础知识补给站】

零基础补给：OOM killer、回收 (reclaim) 与节流 (throttle) 的区别；memory.peak 为何重要。

当内存耗尽时，Linux 的 OOM killer (Out-Of-Memory killer) 登场：它给每个候选进程打分 (oom_score, 大致按内存占用加权, 可用 oom_score_adj 人工调整)，杀得分最高者。cgroup 语境下，组超过 memory.max 会触发「组内 OOM」，只杀该组的进程。OOM 的特点是不可逆、有状态程序损失惨重。

回收 (reclaim) 温和得多：内核把该组占用的页面中「不常用」的换出去或丢掉重读 (页缓存可直接丢, 匿名内存写入 swap)。回收发生在分配路径上：进程申请内存，内核说「先等等，把你家旧报纸卖了再说」，于是分配变慢——这个「变慢」就是节流 (throttle)。memory.high 触发的就是回收 + 节流：内核让超限组的内存分配睡眠一小段时间 (传统上延迟由内核写死)，既施压又不杀人。

memory.peak 则是 v2 提供的一个只读文件，记录该组自创建以来的内存**历史峰值**。它对测量至关重要：容器级每秒采样只能看到「采样瞬间的值」，1-2 秒的突发很可能落在两次采样之间被漏掉；而 peak 是内核逐次记账的最大值，突发再短也逃不掉。AgentCgroup 的 wrapper 在每个工具调用结束后读取临时 cgroup 的 memory.peak 写入日志，等于给每次命令留下了一张不会说谎的「最高水位照片」，这是论文「13.7× 差异」「518 MB P95」这类精确数字的数据来源 (笔者据仓库分析)。

Insight#1 memory.high 与 memory.max 的本质区别是「可逆性」：前者是可以撤销的劝告，后者是不可撤销的处决——论文「优雅降级而非终止」的设计纲领，翻译成内核语言就是「尽量只用 high，万不得已才动 max」。

Insight#2 论文对 high 的批评常被误读为「high 没用」，其实它批评的是「单点 high」：一个阈值对全容器生效才会误伤基线；AgentCgroup 的做法证明 high 本身是对的工具有，只是要用在对的粒度上。

3.3 生命周期三件套：cgroup.freeze / cgroup.kill / memory.oom.group

除了限额，cgroup v2 还提供三个生命周期控制原语，论文第 2 节逐一列出：

- **cgroup.freeze**：向该文件写 1，整个子树的所有进程被冻结 (暂停调度，像按了暂停键)，写 0 解冻。冻结中的进程不占 CPU、状态完整保留。
- **cgroup.kill**：写 1 会向组内所有进程发 SIGKILL。它的价值在于处理「并发 fork」——直接向进程发信号有竞态 (发信号的瞬间进程又 fork 了新进程)，kill 文件由内核保证原子性，一个都跑不掉。
- **memory.oom.group**：置 1 后，组内任何进程触发 OOM 时整组被原子地一起杀掉，避免「杀一半留一半」的半死不活状态 (比如杀掉了 worker 却留下等它的 master)。

论文在设计部分 (第 5 节) 把 freeze 提升为「优雅降级」阶梯的中间档：内存压力上升时，eBPF 程序先按优先级分级执行——用 memory.high 延迟做节流，再用 cgroup.freeze 做冻结，而不是直接终止，从而保住 Agent 积累的状态。这个「节流 → 冻结 → 万不得已才终止」的阶梯是论文资源哲学最凝练的表达。不过需要诚实指出：仓库分析显示，当前开源代码中并没有任何

cgroup.freeze 的写入逻辑 (wrapper、daemon、memcg_controller 三处均无)，冻结档目前属于论文宣称而原型未实现的机制，这一点会在第 10、11 章详细对账。

【零基础知识补给站】

零基础补给：为什么「暂停」比「杀死」对有状态程序重要得多？

传统无状态服务（比如一个 Web 后端）被杀掉的代价很小：请求是无状态的，重启进程、流量重新调度即可。所以传统资源管理体系里「杀」是常规武器——Kubernetes 的 BestEffort Pod 说杀就杀，oomd 的核心动作也是杀。

AI Agent 完全相反，它的状态有三种不可恢复性。其一是**上下文不可外置**：LLM 的对话上下文（几十轮推理记录、中间结论）只活在 Agent 进程的内存里，没有 checkpoint 机制，杀掉即归零。其二是**冷启动昂贵**：论文测得容器 + 框架初始化占任务时延 31–48%（镜像 ID 重映射、Node.js 启动、API 连接），重启等于先交一笔几分钟的税。其三是**执行非确定**：同一任务重跑会走完全不同的解题路径（论文实测执行时间差 1.8×），重跑不保证收敛到原来的方案。这三条合起来，论文称之为 kill-and-restart 的「三重惩罚」。

冻结 (freeze) 恰好绕开全部三重惩罚：进程不丢内存、不用重启、不用重跑，只是让出 CPU 和内存的增长权。这就是为什么 freeze 在论文的降级阶梯中占据关键位置——它是「既要施压又要保命」的唯一原语。理解了这一节，就能明白第 4 章错位分析里「kill acceptable（可杀）是传统工具的隐含假设」这句话的分量。

Insight#1 生命周期三件套揭示了 cgroup 设计者的深意：资源管理从来不只是「限多少」，更是「怎么停」——暂停、原子杀、整组同命运，分别对应可恢复、防逃逸、防半死三种工程需求。

Insight#2 论文把 freeze 写进降级阶梯但原型未实现，这个落差本身就传递了信息：三档阶梯在概念上成立，但「何时冻结、冻结谁、冻多久」的策略细节可能远比论文一句话呈现的复杂（笔者推断）。

3.4 为什么容器 = namespace + cgroup

到这里可以把三块拼图合上了。平时说的「容器」（Docker、Podman、Kubernetes 的 Pod）并不是一种内核对象，而是两个内核机制的**组合产品**：**namespace（命名空间）**负责隔离视野——让容器内的进程看到自己的文件系统、自己的进程号、自己的网络栈，仿佛独占一台机器；**cgroup**负责隔离资源——限制这个容器能用多少 CPU、多少内存。业界有句行话：namespace 让容器「看不见别人」，cgroup 让容器「用不了太多」。

本文实验用的是 Podman（一个与 Docker 兼容、无守护进程的容器引擎），每个 SWE-rebench 任务跑在一个独立 Podman 容器里。论文提到的一个实现细节恰好能说明 namespace 的成本：初始化开销的大头是「Podman 的用户命名空间 ID 重映射（user-namespace ID remapping）」——为了让容器内的 root 映射为宿主机上的普通用户（rootless 模式，更安全），Podman 要给镜像 overlay 文件系统里每一层文件的属主 ID 做一次换算，镜像越大换算越久，所以初始化时间和镜像大小正相关。这解释了为什么 31–48% 的初始化开销不是偶然浪费，而是安全隔离的直接价格（详见第 5 章测量口径）。

对论文论证最关键的是这个推论：既然容器 = namespace + cgroup，那么「比容器更细粒度的资源域」并不需要发明新机制——cgroup 树本来就可以在容器内部继续分叉。AgentCgroup 正是这么做的：不动容器的隔离边界（namespace 层原封不动），只在容器对应的 cgroup 之下挂

出会话级、工具调用级子节点。安全模型不变，控制粒度细了两个数量级。

【零基础知识补给站】

零基础补给：动手感受一下 cgroup——用 shell 创建一个资源受限的「迷你容器」。

在现代 Linux (cgroup v2) 上，下面这组命令就是一个最简资源实验（需要 root）：

```
mkdir /sys/fs/cgroup/demo —— 创建一个叫 demo 的控制组；
echo 500M > /sys/fs/cgroup/demo/memory.max —— 硬上限 500 MB，超了触发组内 OOM；
echo 400M > /sys/fs/cgroup/demo/memory.high —— 软限 400 MB，超了开始回收 + 节流；
echo $$ > /sys/fs/cgroup/demo/cgroup.procs —— 把当前 shell 放进去，之后它 fork 的所有进程都受约束；
cat /sys/fs/cgroup/demo/memory.current —— 实时查看组内存占用；cat
/sys/fs/cgroup/demo/memory.peak —— 查看历史峰值；
cat /sys/fs/cgroup/demo/memory.events —— 查看 high 命中次数、OOM 次数等事件计数。
```

整个过程中没有安装任何软件——cgroup 是内核自带的、以文件为接口的子系统。Docker 创建的每个容器，本质上就是替你自动执行了这样一组操作（外加 namespace 隔离）。AgentCgroup 的 bash wrapper 做的事情与此完全同构：为每条 `bash -c` 命令 `mkdir` 一个临时组、写 `memory.high`、把 PID 写进去、结束后读 `memory.peak` 再 `rmdir`。理解了这组命令，论文第 5 节的机制设计就再也没有神秘感可言。

Insight#1 「容器 = namespace + cgroup」这个分解是全文机制设计合法性的来源：

AgentCgroup 没有发明任何新隔离机制，只是把已有的 cgroup 树往容器内部再延了两层——这意味着它的安全性论证可以直接继承容器的成熟论证。

Insight#2 Podman 的 ID 重映射开销提醒读者：容器技术栈里很多「税」是安全功能而非性能缺陷，论文把初始化 31–48% 列为待优化项，但其中相当一部分要靠镜像层设计（而非资源控制）来省（笔者推断）。

第 4 章 背景（下）：eBPF 深度基础

第 3 章解决了「资源管理的控制点在哪」（cgroup），本章解决「怎么让控制逻辑足够快、足够灵活」（eBPF）。论文第 2 节说，eBPF 让 Linux 得以调和「标准化接口」与「动态工作负载」之间的矛盾，办法是引入**可编程的强制执行点**。本章把这句话彻底展开。

4.1 eBPF 是什么：内核里的安全虚拟机

eBPF（extended Berkeley Packet Filter）通常被描述为「内核里的安全虚拟机」：用户可以把一段小程序（用受限的 C 编写，编译成 BPF 字节码）注入正在运行的内核，挂在某个内核事件上（一次系统调用、一个网络包、一次调度决策），事件发生时这段程序以内核态、全速执行，执行完把结果通过共享数据结构送回用户态。论文引用内核文档^[23] 支撑其安全性论述——文献 [23] 是内核官方文档《eBPF Verifier》，即 eBPF 验证器的设计文档，验证器正是「安全」二字的来源。

「安全」不是口号，而是一套强制检查：任何 BPF 程序在加载前必须通过**验证器（verifier）**——内核用静态分析模拟执行程序的所有路径，确认它必定终止（有指令数上限、循环必须可证明有界）、不越界访问内存、不泄露内核指针、权限匹配。验证不过，拒绝加载。于是历史上「想改内核行为就得写内核模块、一个 bug 拖垮整机」的风险被关进了笼子。这对本文的意义是：资源控制逻辑可以直接住在内核的强制执行点上（微秒级反应），又不需要承担内核模块的风险——响应错位（用户态反应太慢）的解法由此成为可能。

eBPF：把「安全的小程序」动态装进内核，微秒级反应

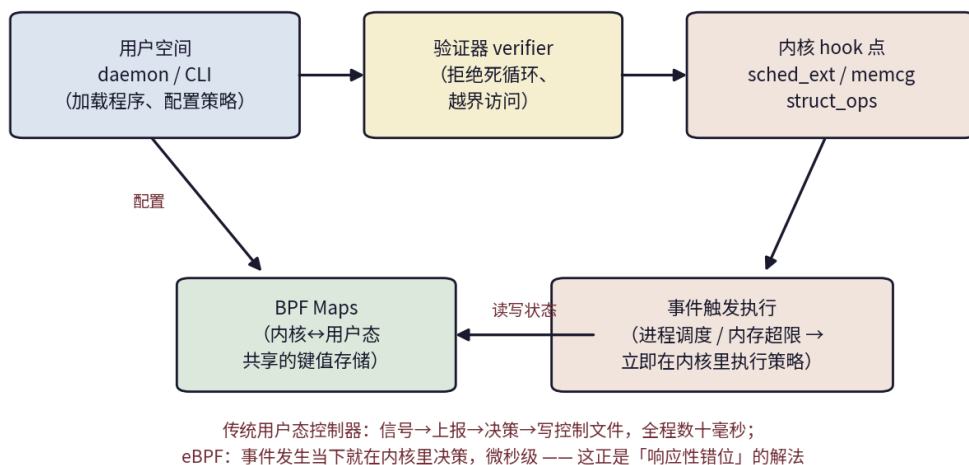


示意图 D 3（本报告绘制）：eBPF 程序的生命周期——C 源码编译为字节码，经验证器检查后 JIT 为本地机器码，挂在内核挂点上，通过 BPF Maps 与用户态交换数据

怎么读这张图：从左到右是「一条 BPF 程序的一生」。左侧用户态：开发者写受限 C，clang 编译成 BPF 字节码，由 libbpf 库加载进内核。中间内核态：验证器先做静态安全检查，通过后 JIT（即时编译）成本地机器码，然后 attach（挂接）到某个挂点上——图中画了三种：tracepoint（内核预定义的稳定埋点）、kprobe（任意内核函数入口的动态探针）、struct_ops（替换内核

结构体的回调函数，本文的 `sched_ext` 和 `memcg_bpf_ops` 都属于此类）。右侧是 BPF Maps：内核与用户态共享的键值存储，是双向通信的通道——AgentCgroup 的 daemon 通过它下发策略参数（写 `.bss` 配置或 map），BPF 程序通过它上报统计。

【零基础知识补给站】

零基础补给：eBPF 简史——从 tcpdump 到「内核的 JavaScript」。

故事始于 1992 年的 BPF（Berkeley Packet Filter）：tcpdump 抓包时，用户只想看某类包，如果每个包都拷贝到用户态再过滤就太慢了，于是 BSD 在内核里放了一台微型虚拟机，用户上传一段过滤小程序，在内核里直接对包做「留/丢」判断。这就是 BPF 的全部初心：**把决策下沉到数据所在的地方**。

2014 年，Alexei Starovoitov 主导的 eBPF 把它推广为通用内核虚拟机：寄存器从 2 个扩到 11 个，增加了 Maps、辅助函数、尾调用，挂点从网卡扩展到系统调用、调度器、文件系统……今天 eBPF 被称为「内核的 JavaScript」——就像 JS 让网页可编程而不必改浏览器，eBPF 让内核可编程而不必改内核源码、不必重启。可观测性（Cilium、Pixie）、网络（Cilium 负载均衡）、安全（Tetragon）是它最成功的三个战场，Facebook/Meta、Google、Netflix 都在生产环境大规模使用。本论文则代表了第四个方向的探索：用 eBPF 做**资源控制**——观测是读，控制是写，写的风险大得多，所以 `sched_ext` 和 `memcg_bpf_ops` 这类「可编程控制点」出现得晚、进主线也更谨慎。理解了这段历史，就能理解为什么第 4.3 节的 `memcg_bpf_ops` 至今还是 RFC 补丁：内核社区对「让用户程序干预内存管理」天然警惕。

Insight#1 eBPF 的核心思想三十年没变——「把决策下沉到数据所在的地方」；变的是「数据」的含义，从网络包扩展到调度事件、内存事件。AgentCgroup 是把这套思想搬到 Agent 资源管理的第一批工作。

Insight#2 验证器的存在解释了论文为什么敢让控制逻辑进内核：安全不是靠开发者小心，而是靠加载时的数学级检查——这也是「内核内强制」能被系统社区接受为正当手段的前提。

4.2 sched_ext：用 BPF 写 CPU 调度器

CPU 侧的可编程接口是 `sched_ext`，论文引用内核文档^[15]——文献 [15] 是 Tejun Heo、David Vernet、Josh Don 撰写的内核文档《Extensible Scheduler Class》。`sched_ext`（2024 年合入 Linux 6.12 主线）允许用 BPF 程序实现整个 CPU 调度器：内核定义了一组回调（哪个任务该上 CPU、运行多久、任务唤醒时放哪个队列……），开发者用 BPF 实现这些回调，就得到了一个可热插拔的自定义调度器。Meta 用它在大规模生产环境试验游戏服务器、Web 服务的定制调度策略。

`sched_ext` 设计中最关键的是 **fail-safe 回退**（论文原话：with fail-safe reversion to default behavior on errors）：如果 BPF 调度器出了任何错——程序崩溃、验证失败、心跳超时——内核会立即把所有任务切换回默认的 CFS/EEVDF 调度器，系统继续正常运行。调度器是内核里最不能坏的部件（坏了整机卡死），fail-safe 意味着「BPF 调度器最多浪费你一次部署，绝不会弄挂你的机器」。这也是 `sched_ext` 能被内核社区接受的核心理由。

AgentCgroup 用的是内核源码树里自带的示例调度器 `scx_flatcg` 的移植版（仓库分析确认文件头完整保留了上游注释）。`flatcg` 的策略是「扁平化层级权重」：把 cgroup 树的权重沿层级折叠复合（例如 session 权重 $150 \times$ 组内份额），压平成单层后按虚拟时间（vtime）做公平调度

——「先选 cgroup，再选组内任务」的两级结构。daemon 只需给 session_high 写 `cpu.weight=150`、session_low 写 50，3:1 的 CPU 优先级就由调度器在每次调度决策中自动执行，反应速度是调度器级的纳秒/微秒尺度，不存在用户态轮询。

【零基础知识补给站】

零基础补给：CPU 调度器到底在调度什么？什么是权重公平？

一颗 CPU 核心同一时刻只能跑一个线程，调度器的工作就是决定「下一个跑谁、跑多久」。Linux 默认调度器（CFS/EEVDF）的思想是「完全公平」：给每个可运行线程记账（虚拟运行时间 `vtime`），总是挑欠账最少的跑，长期看人人均分。但「人人均分」在多租户场景恰恰是错的——付费高的租户理应多得，于是引入**权重**（`weight`）：`vtime` 的增长速度除以权重，权重 150 的组记账速度只有权重 50 的组的 1/3，于是它被选中的频率就是后者的 3 倍。这就是 cgroup 的 `cpu.weight` 的语义，比例而非绝对值才有意义。

传统上调度策略（多少权重、多大时间片、如何抢占）是内核写死的，管理员只能调参数。`sched_ext` 把策略本身变成了可替换的软件：想让「延迟敏感的任务永远插队」？写 200 行 BPF 就行，不用改内核、不用重启、跑挂了自动回退。对本文的场景，价值在于：Agent 会话的 CPU 优先级（高优先级租户 3:1 碾压低优先级）是在每次上下文切换时由内核内逻辑执行的，亚秒级的工具突发完全逃不出它的视线——对比之下，用户态调权重的方案（定时脚本改 `cpu.weight`）反应是秒级的，突发早就结束了。还要指出一个仓库分析发现的隐式耦合：启用 `sched_ext` 后，Docker/K8s 的 `cpu.max` 带宽配额不再自动生效（设计文档 `design_v3.md` 明确记录了这一点），这是用 BPF 接管调度的真实代价。

Insight#1 fail-safe 回退是 `sched_ext` 对系统社区的政治承诺：可编程调度可以试验、可以失败，但绝不危及整机——这条设计原则（机制可败、系统不塌）值得所有可编程内核功能效仿。

Insight#2 论文用现成的 `scx_flatcg` 而非自研调度器，是诚实的工程选择：本文的 CPU 贡献不在调度策略创新，而在「把 cgroup 权重机制接进了 Agent 会话模型」；期待调度策略本身针对 Agent 负载优化，是后续工作的空间（笔者推断）。

4.3 memcg_bpf_ops：内存控制器的 BPF 钩子

内存侧的可编程接口是 `memcg_bpf_ops`，论文引用^[41]——文献 [41] 是 Hui Zhu 发表在 LWN.net 的内核补丁系列《mm: memcontrol: Add BPF hooks for memory controller》（RFC PATCH bpf-next，论文文本记为 2026 年 1 月 23 日的 v3，共 12 个补丁；仓库随附的 `patches.mbox` 已是 v6）。它为内存控制器（memcg）定义了一组 BPF `struct_ops` 回调，其中本论文最关键的是 `get_high_delay_ms`：当某个 cgroup 的内存突破 `memory.high` 进入节流时，内核不再使用写死的延迟，而是调用 BPF 程序询问「这个组该睡多少毫秒」——节流力度从此可编程。另一个回调 `below_low` 让 BPF 程序可以声称「这个组应被视为低于 `memory.low`」，从而临时豁免回收。

必须对读者强调：**这是一个尚未合入主线的 RFC 补丁**。RFC（Request for Comments）意味着作者把它贴到邮件列表征求意见，接口随时可能改。直接后果是：要运行 AgentCgroup 的内存侧强制，必须用 bpf-next 内核源码加上这 12 个补丁**自行编译内核**（论文评估用的是 6.19.0-rc5 + 补丁）；README 里「Linux 6.12+」的前提只覆盖 `sched_ext`，不覆盖内存侧。仓库为此提供了折中方案：`memcg_controller.py` 实现了一个纯用户态的 cgroup v2 回退后端，用写

memory.low/high 文件模拟近似语义（实测只能达到约 $1.85\times$ 的节流区分度，远弱于 BPF 版）。这段「依赖未合入特性」的历史与局限，第 10、11 章还会展开。

memcg_bpf_ops 在 AgentCgroup 里的具体用法（依据仓库源码）：挂 HIGH 优先级 cgroup 的 below_low 回调——当 BPF 程序检测到 HIGH 组缺页速率超标（1 秒窗口内 page fault 计数超过阈值）时进入 1 秒「保护窗口」，窗口内让内核把 HIGH 组当作低于 memory.low 处理，跳过对它的回收；挂 LOW 组的 get_high_delay_ms——窗口内 LOW 组一旦超过自己的 memory.high（实验中故意设低，400 MB），每次分配被罚睡指定毫秒数（实验用 50 ms）。一保一压之间，实现了「高优先级 Agent 的内存需求永远优先被满足」。

【零基础知识补给站】

零基础补给：这个补丁到底改了内核的哪根筋？

内存管理是内核最古老、最保守的子系统，「让用户代码干预回收与节流」在历史上是不可想象的。memcg_bpf_ops 的突破点选得很刁：它不动回收算法本身，只开放了两个「询问点」。

第一个询问点在节流路径上。传统行为是：组超 memory.high $\rightarrow$ 内核让分配睡眠一个固定时长（按超限程度计算的写死公式）。补丁改成： $\rightarrow$ 内核先问 BPF「睡多久？」，BPF 可以根据这个组是谁（高优还是低优）、当前谁在承压，返回 0（不罚）到任意毫秒数。第二个询问点在回收路径上：内核选择回收对象时会跳过低于 memory.low 的组，补丁让 BPF 也能回答「算它低于 low 吗？」，等于给特定组发一张临时「免回收金牌」。

为什么社区谨慎？因为回答这些问题的代码运行在内存管理的热路径上：答得慢，全机分配变慢；答得错，内存被无限豁免可能拖垮整机。验证器只能保证程序「不崩」，保证不了「答得对」。这就是为什么它从 v3 迭代到 v6 仍在 RFC 阶段。对 AgentCgroup 论文，这既是风险也是贡献定位：它是 memcg_bpf_ops 目前最具体的真实用户之一，论文和仓库相当于为这个内核提案提交了一份「杀手级应用场景」的论据——系统研究中「用真实需求牵引内核特性」的案例历来是双赢。

Insight#1 get_high_delay_ms 把「节流」从常量变成函数，这是资源控制史上的范式事件：惩罚第一次可以按「身份」而非仅按「行为」来定——多租户优先级的内核级表达由此成立。

Insight#2 依赖未合入内核特性是本文最大的落地障碍，但反过来，AgentCgroup 为这个 RFC 补丁提供了最有说服力的应用场景——论文与内核补丁之间存在互相成就的关系（笔者推断）。

Insight#3 用户态回退后端 $1.85\times$ vs BPF 版的效果差距，量化了「内核内强制」的不可替代性：不是用户态做不到，而是用户态做不到微秒级、做不到分配路径内。

4.4 BPF CO-RE 与 libbpf：原型为什么选这套工具链

论文第 5 节末尾写道：原型用 C 语言基于 libbpf 和 BPF CO-RE 实现^[24]——文献 [24] 是内核官方文档《libbpf Overview》，介绍 libbpf 这个 BPF 应用开发的官方用户态库。这句话对不熟悉 BPF 工程生态的读者可能只是一笔带过，但它其实解释了原型代码形态的一切。

libbpf 是内核源码树维护的官方 C 库，负责把编译好的 BPF 字节码加载进内核：解析 ELF、创建 Maps、触发验证、完成挂接。与之配套的工程惯例是 **skeleton（骨架）**：构建时由 bpftool 把 BPF 目标文件预处理生成一个 C 头文件，用户态程序像访问普通结构体一样访问 BPF 程序的 Maps 和全局变量（比如直接写 skel->bss->local_config 下发配置——AgentCgroup 的 memcg_priority loader 正是这么做的）。

CO-RE (Compile Once, Run Everywhere, 一次编译处处运行) 解决的是 BPF 工程化的最大痛点：BPF 程序经常要读内核结构体的字段（比如从 `task_struct` 里取进程信息），而结构体布局随内核版本变化——在 6.8 上编译的程序到 6.15 上字段偏移可能就错了。CO-RE 的办法是：编译时保留字段的「名字」而非「偏移」（借助 BTF，内核自带的类型信息），加载时由 `libbpf` 对照当前内核的 BTF 做**重定位**，自动修正偏移。于是一份编译产物可以在不同版本的内核上直接运行。对 AgentCgroup 这种需要用户自编译内核 + 补丁的项目，CO-RE 意味着至少 BPF 程序本身不必跟着内核重编——原型选它是工程上的必然。

【零基础知识补给站】

零基础补给：三种挂点有什么区别？`struct_ops` 为什么是新范式？

BPF 程序必须挂在某个「点」上才会被执行，常见三类挂点性质迥异。

tracepoint (跟踪点)：内核开发者预先埋下的稳定埋点（如 `sched_process_exec` 表示「进程 `exec` 了」），跨版本相对稳定，只能读事件参数，不能改变内核行为——纯观测。AgentCgroup 的进程监控组件用的就是它（外加一个挂在 `bash readline` 上的 `uretprobe`，即用户态函数返回探针）。

kprobe (内核探针)：动态挂到几乎任何内核函数的入口/出口，极其灵活，但函数名和签名随版本漂移，稳定性差——适合调试和临时观测，不适合长期产品。同样以读为主。

struct_ops (结构体操作)：2020 年引入的新挂点类型，也是真正的范式转变——内核把某个子系统的「策略结构体」（一串函数指针）开放出来，BPF 程序可以实现其中若干回调，相当于「用 BPF 写半个内核模块」。前两者是「内核做事时通知你」，`struct_ops` 是「内核做事时问你怎么办」：`sched_ext` 问「下一个跑谁」，`memcg_bpf_ops` 问「超了 `high` 睡多久」。观测与控制的楚河汉界就在这里。

`struct_ops` 的挂接方式也特殊：它不是一个程序挂一个点，而是一张特殊 Map（装着函数指针表）整体 `attach` 到目标子系统上，还可以相对某个 `cgroup fd` 挂载（AgentCgroup 的两个内存回调正是按 `cgroup` 分别挂载的）。理解 `struct_ops`，就理解了为什么说 eBPF 正在从「内核的探针语言」进化为「内核的策略语言」——本论文正是这个进化在资源管理方向上的路标之一。

Insight#1 从 `tracepoint` 到 `struct_ops` 的演进，本质是内核从「被观测对象」变成「被编程对象」——AgentCgroup 的两个核心机制（`sched_ext`、`memcg_bpf_ops`）恰好都是 `struct_ops`，这不是巧合而是时代位置。

Insight#2 CO-RE + skeleton 让 BPF 工程体验接近普通 C 开发，是原型能在「C 2,933 行 + 内核补丁」的体量内完成的关键基础设施红利；五年前做同样的系统，工程量至少翻倍（笔者推断）。

第 5 章 测量方法：数据是怎么来的

测量驱动设计论文的可信度，七分靠方法。本章把论文第 3.1 节的实验设置拆到螺丝钉级别：在什么机器上跑、跑什么任务、用什么模型、数字怎么量出来的。读完本章，第 6–8 章的每一个结论你都能自己判断该信几成。

5.1 实验平台与任务集

所有表征实验跑在**单台机器**上：Intel Core Ultra 9 285K（24 核）、128 GB DDR5 内存、Ubuntu 24.04、Linux 内核 6.15.11、cgroup v2。单机设置意味着结论描述的是「单机多租户密度」问题，不涉及跨机调度——这是合理的第一步，也是外推到数据中心时要注意的边界（单机结论关于「内存是瓶颈」在集群层面大概率仍成立，但数值会变，笔者推断）。

任务集来自 SWE-rebench 基准^[5, 16]，共 144 个任务：GLM 模型跑 111 个，Haiku 模型跑 33 个，**且 33 个 Haiku 任务是 111 个 GLM 任务的子集**，所以所有跨模型对比都在这个共享交集上进行——这个细节保证了模型对比不受任务难度差异污染，是方法上的加分项。每个任务使用 SWE-rebench 官方 Docker 镜像，经过去重后共 114 个镜像，单个 2.9–17.3 GB，中位数 3.5 GB（论文图 4(a) 给出分布），111 个 GLM 任务的镜像合计 456 GB。镜像平均 3.5 GB 这个数字本身后来成了结论之一：是典型微服务镜像的 7 倍、serverless 函数的 70 倍。

对照组方面，论文把测量结果与三类已有负载研究的公开数据对比：Azure serverless 函数^[30]（文献 [30] 是 USENIX ATC 2020 的《Serverless in the Wild》，对大型云厂商 serverless 负载的开创性测量，本文用它提供 FaaS 维度的对照数据）、Azure 虚拟机/微服务^[9]（文献 [9] 是 SOSP 2017 的《Resource Central》，微软 Azure 内部负载的测量与预测系统，本文的对照基准之一）、Google 批处理集群 Borg^[32]（文献 [32] 是 EuroSys 2015 的 Borg 集群管理论文，代表批处理/HPC 负载）。第 6 章会详细展开这些对照。

【零基础知识补给站】

零基础补给：为什么测量论文要交代「在什么机器上跑」？

系统测量和物理实验一样讲究「实验条件可复现」：CPU 型号决定单核性能与核数（24 核意味着「CPU 平均利用率 13%」离饱和很远，结论「CPU 不是瓶颈」在这台机器上成立）；内存容量决定「能塞多少实例」的算术（128 GB ÷ 峰值 2–4 GB = 32–64 实例这个推算直接来自这台机器的配置）；内核版本决定可用的 cgroup/eBPF 特性（6.15.11 很新，包含 sched_ext，且有 cgroup v2 完整语义）。

单机 vs 集群也是个重要的方法论选择。单机测量排除了网络、调度器、镜像分发等变量，让「负载本身的资源行为」成为唯一的观测对象——先在小而干净的环境里把现象钉死，再谈大规模，是谨慎的做法。但读者要警惕两类外推错误：其一，单机 24 核上「CPU 不是瓶颈」，换成 4 核的云主机就不一定（事实上论文第 6 节的评估机器就只有 4 核）；其二，单机内存 128 GB 时「内存是瓶颈」的密度上限，在集群层面还要叠加大镜像分发、冷启动排队等新瓶颈。本报告在第 11 章会系统讨论这些外部效率问题。

Insight#1 「33 是 111 的子集」这个设计细节价值被低估：它让跨模型对比成为配对比较（paired comparison），排除了任务混合差异——很多 Agent 测量工作输就输在对比口径上。

Insight#2 456 GB 镜像、144 个任务、单机 24 核——这个实验规模在个人实验室条件下已接近极限，也解释了为什么表征只覆盖一个框架一个基准：不是不想，是成本封顶了（笔者推断）。

5.2 两个模型、两种部署形态：云 API vs 本地 GPU

论文用了两个底层模型，构成一组精心选择的对照：**Claude Haiku 4.5**（Anthropic 的轻量模型，通过云端 API 调用）与 **GLM-4.7-Flash**（智谱的开源模型，在本地 GPU 上推理）。Agent 框架统一用 Claude Code^[1]，所以两个模型的差异只来自「大脑」本身及其部署形态。

为什么这组对照有意义？因为部署形态本身就是资源画像的一部分。云 API 模式下，LLM 推理发生在远端，本地 CPU 只做响应解析、上下文管理等杂活；本地 GPU 模式下，推理的算力负担转移到本地 GPU，本地 CPU 在工具调用之外也持续有负载。论文的数据精确呈现了这种分化：

Haiku 平均 CPU 利用率 13.2%，8.2% 的采样点 CPU 超过 50%；GLM 平均只有 7.6%，仅 0.5% 的采样点超过 50%——同任务上有 $1.7\times$ 的 CPU 利用率差异。行为模式也不同：Haiku 会把工作分派给子 Agent（sub-agent 调用占其工具时间的 17.3%）并使用 Web 搜索，GLM 则几乎把全部计算集中在本地 Bash 调用（占其工具时间 98.8%）。

这个对照给「不可预测性」结论提供了最硬的一条证据：连「换哪个模型」都会显著改变资源画像（ $1.7\times$ CPU 差异），那么任何针对单一模型调好的资源策略，在模型升级或更换时都会失效。论文把它列为适配错位的第三条论据（资源需求「跨模型进一步分化」）。

【零基础知识补给站】

零基础补给：云端 API 与本地推理，资源账怎么算？

用云 API 跑 Agent，本地这台机器的账本上只有：Agent 框架进程（Node.js，约 185 MB 基线）、网络收发、JSON 解析、以及所有工具子进程。LLM 推理的 GPU 成本发生在云厂商的数据中心，本地不可见——所以论文测的「Haiku 的 OS 资源」其实是「Agent 框架 + 工具」的资源，不含模型本身。

用本地 GPU 模型跑，账本变了：GPU 和显存被推理占用（这部分在论文的 CPU/内存采样口径之外，因为采样的是容器内进程），CPU 上则有推理框架的调度、token 化等持续开销，而且「LLM 思考」阶段不再等于「本地空闲」——GLM 数据里 LLM 推理阶段也有稳定 CPU 负载，这解释了为什么 GLM 的 CPU 突发归因更分散（只有 30.2% 的 CPU 突发落在工具调用内，Haiku 为 55.3%）。对云厂商而言这组对照的现实含义是：托管「自带大脑」的 Agent（客户本地模型）和托管「租用大脑」的 Agent（调云 API）是两种不同的资源生意。前者要规划 GPU，后者 GPU 压力为零但网络与解析开销更高。论文用同一框架、同一批任务把两种形态的画像都测了出来，这是它对基础设施规划者最实用的输出之一。

Insight#1 这组对照的深层贡献是把「模型选择」正式确立为资源画像的独立变量——此前社区默认换模型只影响效果与价格，本文证明它还以 $1.7\times$ 的幅度影响 OS 资源需求。

Insight#2 两个模型都是「轻量款」（Haiku、Flash 均为各自家族的小模型），这其实是贴近生产的选择：编程 Agent 的主力恰恰是便宜快速的小模型，而非旗舰模型——测量对象选得务实（笔者推断）。

5.3 测量口径：每个数字是怎么量出来的

论文第 3.1 节交代了测量口径，每一条都值得细读，因为口径决定结论的边界：

- **资源采样**：对每个任务，每秒（1 秒间隔）采样一次 CPU 利用率与内存用量（CPU 归一化到单核，100% = 一个核跑满）；同时记录每次工具调用的类型与时间戳。仓库分析补充了实现：宿主机脚本 `run_swebench.py` 每秒采集容器级 mem/CPU 写入 `resources.json`。
- **工具时间定义**：从「LLM 发出 tool-use 请求」到「对应结果返回」为止，包含框架分发、进程执行、结果收集三段。注意这个口径比「子进程墙钟时间」略大，包含了框架开销——这是从用户体验出发的诚实口径。
- **初始化构成**：分两阶段——Podman 用户命名空间 ID 重映射（dominant，随镜像大小增长）+ Agent 框架启动（Node.js 进程与 API 连接）。**镜像拉取时间被排除在外**，否则初始化占比会更难看。
- **框架注入方式**：Claude Code 框架及其依赖从宿主机 **bind-mount（绑定挂载）** 进每个容器，消除了逐容器安装的开销——这也让「框架基线约 185 MB」在 144 个任务间严格可比（Haiku 183 MB、GLM 188 MB）。
- **per-tool-call 峰值**：论文正文未强调、但仓库分析显示，启用 wrapper 时（`--enable-wrapper`），`bash_wrapper.sh` 会为每次工具调用建临时 cgroup 并读取 `memory.peak`——「13.7×」「518 MB P95」这类工具级精度数字来自这个内核记账值，而非每秒采样（笔者据仓库分析）。

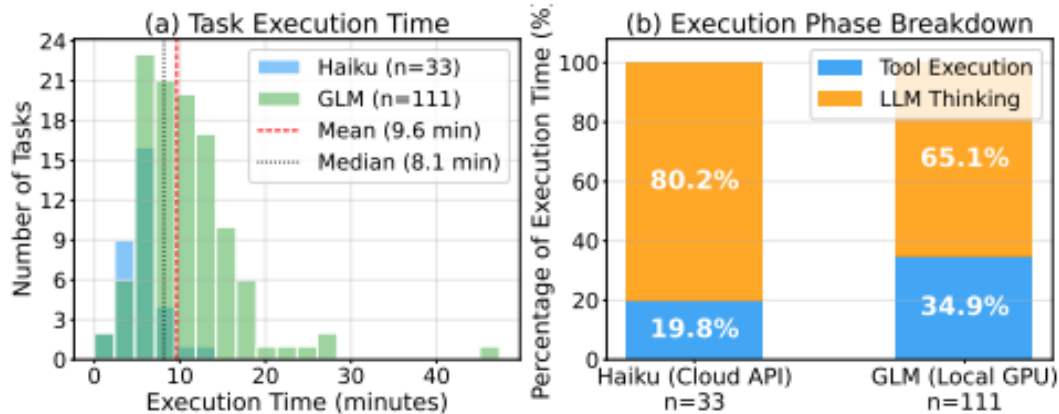


图 1（论文原图）：(a) 任务执行时间分布，GLM 平均 10.8 分钟、Haiku 平均 5.8 分钟、总中位 8.1 分钟；(b) 执行阶段划分——活跃时间内工具执行占 19.8%（Haiku）/ 34.9%（GLM），其余为 LLM 思考

怎么读这张图：左图 (a) 是两个模型任务时长的分布直方图，注意 GLM（本地小模型）反而比 Haiku（云端模型）慢近一倍——思考速度是一方面，GLM 更依赖重试（第 7 章会讲它平均 3.9 个重试组/任务）是另一方面。右图 (b) 是把「初始化」剔除后，活跃时间内在「工具执行」与「LLM 思考」之间的切分：两根柱子分别归一化到 100%，读法是比较色块比例而非柱高。这张图是全文的第一个证据：如果 LLM 思考只占活跃时间的 65–80%、再算上 31–48% 的初始化，那么「Agent 慢」的账有一半以上要记在非 LLM 因素头上。

【零基础知识补给站】

零基础补给：分位数、变异系数、bind mount、ID remapping，以及采样混叠风险。

P50 / P95 / P99（分位数）：把一组数据从小到大排序，P95 是「排在 95% 位置的那个值」，含义是「95% 的情况不超过它」。它比均值耐极端值干扰，是描述时延、峰值这类「长尾分布」的标配。本文用 P95 描述测试类命令的内存尖峰（518 MB），意思是「跑测试时有 5% 的概率比这还高」。

CV（变异系数，Coefficient of Variation） = 标准差 ÷ 均值，用于比较不同量纲数据的离散程度。论文测得跨任务峰值内存 CV = 147%——作为参照，一般认为 CV 超 100% 就属于极度分散，「不可预测」结论的统计学外壳就是它。

bind mount（绑定挂载）：把宿主机的一个目录原样映射进容器，容器内看到的是同一份文件。论文用它把 Claude Code 注入每个容器，免去了 144 次安装。

user namespace ID remapping：rootless 容器的安全机制，容器内 UID 0（root）映射为宿主主机上的普通 UID；代价是镜像每层文件的属主 ID 都要换算一遍，镜像越大越慢——这是初始化开销的大头。

采样混叠（笔者推断）：论文测量「突发持续 1-2 秒」，而采样间隔恰好也是 1 秒。按采样定理，要忠实还原一个信号，采样率至少要是其最高变化频率的两倍——1 秒采样对 1-2 秒的突发只是勉强够用：突发的真实峰值可能落在两次采样之间被低估（好在新内核的 memory.peak 记账补上了这一课，wrapper 采集的 per-tool-call 峰值不受此限），突发的起止时刻也可能各偏移最多 1 秒。读论文突发相关数字时应带上这个误差棒。

Insight#1 测量口径里最能体现实验品味的是「工具时间从 LLM 发出请求算起」：它把框架分发开销诚实计入工具侧，避免美化「OS 占比」——而 55-60% 这个耸动数字恰恰因此更可信。

Insight#2 排除镜像拉取时间是必要的口径净化（否则初始化占比会被网络速度主导），但也提醒读者：生产环境中拉取是真实成本，论文的 31-48% 初始化占比是生产值的下界（笔者推断）。

Insight#3 容器级每秒采样 + wrapper 级 memory.peak 记账，构成了「宏观曲线 + 微观峰值」的双层测量——后者是论文工具级结论（13.7×、518 MB）精度的真正来源，也是测量设计与系统设计互相渗透的例证（笔者据仓库分析）。

第 6 章 发现一：执行模型与时间构成

论文的第二个贡献，是把 AI 编程 Agent 的工作负载（workload）当作一种全新的操作系统负载来「画像」。在此之前，学术界对 serverless（无服务器计算）、微服务（microservice）、批处理（batch）负载都做过大规模测量——例如 Microsoft 对 Azure 无服务器函数的测量研究^[30]（Serverless in the Wild, ATC 2020，本文用它作为「短任务」负载的对照基准）、对 Azure 虚拟机的 Resource Central 研究^[9]（SOSP 2017，本文用它作为微服务负载的对照）、以及 Google 对 Borg 集群的测量^[32]（EuroSys 2015，本文用它作为批处理负载的对照）。而本文是首个系统刻画 Agent 沙箱内 OS 级资源动态的工作。本节对应论文 § 3.2，回答一个问题：一个编程 Agent 任务的时间到底花在了哪里？

6.1 任务时长：5–11 分钟，介于 serverless 与批处理之间

论文在 SWE-rebench 基准^[5]上测量了 144 个真实软件工程任务。SWE-rebench 是 Nebius 团队 2025 年提出的自动化基准管线（arXiv:2505.20411），它持续从 GitHub 收集真实 issue 并做「去污染」处理，本文的全部测量都建立在它的任务与官方容器镜像之上。实验使用 Claude Code 框架，底层模型有两个：云 API 的 Claude Haiku 4.5（33 个任务）和本地 GPU 部署的 GLM-4.7-Flash（111 个任务）。

第一个数字就很说明问题：**每个 Agent 任务运行 5–11 分钟**（GLM 均值 10.8 分钟，Haiku 均值 5.8 分钟，全样本中位数 8.1 分钟）。这个时长处在一个尴尬的「无人区」：它比 serverless 函数调用（100 毫秒–2 秒）长几个数量级，又比动辄数小时的批处理作业短得多。时长的跨度直接决定了资源管理策略的适用性——为毫秒级函数设计的冷启动优化、为小时级作业设计的调度器，在这里都用不上。

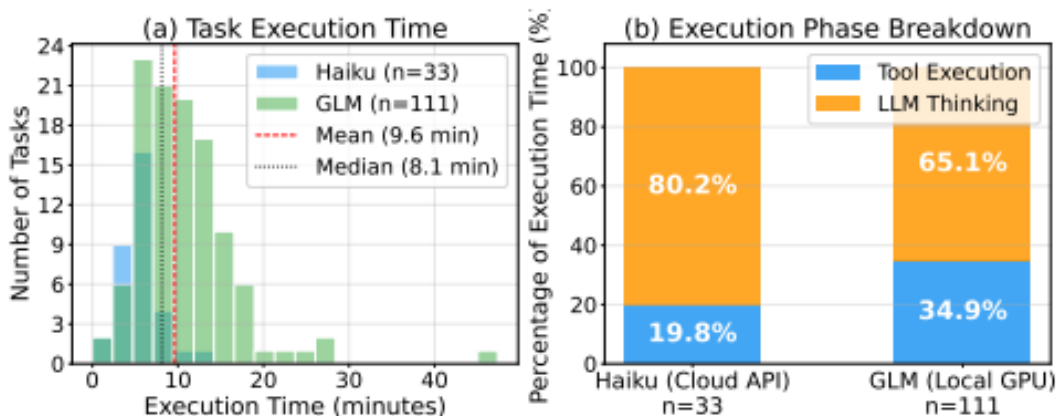


图 1（论文原图）：任务执行时间分布 (a) 与执行阶段划分 (b)。

怎么读图 1(a)（左侧直方图）：横轴是任务执行时间（分钟），纵轴是任务个数。浅色柱是 Haiku 的 33 个任务，深色柱是 GLM 的 111 个任务。可以看到整体分布大致集中在 5–15 分钟区间，均值线 9.6 分钟、中位线 8.1 分钟。注意 GLM 任务明显偏右（本地小模型推理慢、轮次多），Haiku 偏左——模型本身决定了任务节奏。

怎么读图 1(b)（右侧堆叠条）：两根 100% 归一化的条形分别代表 Haiku 与 GLM，把「active time」（刨去初始化后的有效执行时间）切成 LLM Thinking（LLM 推理）与 Tool Execution（

工具执行) 两段。Haiku 的工具段占 19.8%，GLM 的工具段占 34.9%——同一框架、同一批任务，工具占比却差了将近一倍，这个差异的来源我们会在 6.2 小节展开。

【零基础知识补给站】

什么是「任务时长分布直方图」？把 144 个任务按完成时间分桶（例如 0-5 分钟一桶、5-10 分钟一桶），每桶里数有多少个任务，画成柱子，就是直方图。它能让你一眼看出「大多数任务多久跑完」以及「有没有长尾」。论文中的均值（mean）是所有任务时长加起来除以任务数，中位数（median）是排好序后正中间那个数；当中位数明显小于均值时（8.1 对 9.6），说明分布有右偏的长尾——少数特别慢的任务把均值拉高了。为什么时长重要？因为操作系统管理资源的方式强依赖于「任务活多久」：对于只活 1 秒的函数，最划算的做法是提前预热好一堆实例等着；对于活几小时的批作业，值得花时间精细调度、排队优化。而 5-11 分钟的 Agent 任务两头不靠：预热容器代价太高（光初始化就要几分钟），又不像批作业那样值得排队等待。这正是论文后文「三大错位」的第一个伏笔。顺带一提，SWE-rebench 这类基准里的每个任务都是「给你一个真实 GitHub 仓库和一个 issue 描述，请修好它」，Agent 需要自己读代码、改代码、跑测试——这就是它为什么动辄要跑十分钟。

Insight#1 Agent 任务的 5-11 分钟时长不是偶然的中间值，而是「任务复杂度 × 模型能力」共同决定的稳态：模型越弱，轮次越多，任务越长。

Insight#2 GLM 比 Haiku 慢近一倍（10.8 vs 5.8 分钟），说明「任务时长」本质上是模型的函数，而非任务的固有属性——这对任何想按任务类型预估时长的系统都是坏消息。

Insight#3 5-11 分钟恰好落在现有云基础设施的调度盲区里：比预热池能忍受的上限长，比值得排队优化的下限短。这意味着 Agent 平台无法直接复用任何一类现成的调度基础设施。

6.2 OS 层吃掉 55-60%：钱花在 GPU 上，时间花在 OS 上

把任务的端到端时延（latency）拆开，论文得到三组数字：LLM 推理（reasoning）占 40-45%，容器与 Agent 框架初始化占 31-48%，工具执行占 active time 的 20-35%（Haiku 19.8%、GLM 34.9%；全生命周期口径下工具执行占 13-24%）。把「初始化 + 工具执行」这两块都属于操作系统层面（OS-level）的开销加起来：**OS 层吃掉了用户感知完成时间的 55-60%**。换句话说，一个号称「AI」的系统，一大半时间并没有在「思考」，而是在等容器启动、等 Node.js 进程就绪、等 pytest 跑完。

初始化开销值得单独一说。它包含两个阶段：先是 Podman 对容器镜像 overlay 层做用户命名空间（user namespace）ID 重映射——这一步是大头，随镜像大小增长；然后才是 Agent 框架自身启动（Node.js 进程 + 建立 API 连接）。论文特意把镜像拉取（pull）时间排除在外，所以这 31-48% 纯粹是「起容器 + 起框架」的成本，在镜像已经缓存的机器上依然存在。

为什么 GLM 的工具占比（34.9%）反而比 Haiku（19.8%）高？直觉上本地 GPU 推理应该让「思考」更快才对。原文给出的线索是两个模型策略不同：Haiku 会把工作分派给 sub-agent（子 Agent）和 Web 搜索，而 GLM 几乎把所有计算集中在本地 Bash 调用里。在此基础上笔者认为（笔者推断）：GLM 的本地 GPU 推理速度快、单轮思考时间短，导致分母（active time）中 LLM 段缩小，工具执行的相对占比自然被放大；而 Haiku 走云 API，网络往返与长推理让 LLM 段膨胀，把工具占比压低了。这提示我们：**工具时间占比是「思考快」的镜像指标**——推理越快，OS 层

开销在总时延中就越扎眼。

【零基础知识补给站】

什么是 LLM 推理时间、什么是工具执行时间？一个编程 Agent 的工作节奏是「想一步、做一步」的循环：LLM 生成一段文字（可能是思考，也可能是一个工具调用请求，比如「运行 pytest」），这是推理时间；框架收到请求后真正去操作系统里启动进程、执行命令、把结果收回来，这是工具执行时间。论文的计时口径是：从 LLM 发出 tool-use 请求到对应结果返回为止，中间包括框架分发、进程执行、结果收集三段。再看「初始化」：Docker/Podman 容器不是免费启动的。容器镜像由多层文件系统叠加（overlay）而成，为了安全，Podman 要把镜像里每个文件的用户 ID 映射到宿主机的一个隔离区间——镜像越大，文件越多，映射越慢。Agent 用的镜像平均 3.5 GB（下一章详述），所以这一步动辄几十秒到几分钟。之后还要启动 Claude Code 的 Node.js 运行时、连上模型 API，才算「Agent 醒了」。把这些加在一起，任务真正开始干活之前，三分之一左右的时间已经花掉了。理解这个拆分的意义在于：如果你想让 Agent 平台更快，只换更快的 GPU 只能优化那 40-45%；剩下的 55-60% 完全是操作系统问题——这正是这篇论文作为一篇系统论文（而非 AI 论文）的立足点。

Insight#1 55-60% 的 OS 层占比意味着：Agent 基础设施的优化主战场不在 GPU，而在容器启动、进程管理与文件系统——这是传统系统研究的主场。

Insight#2 31-48% 的初始化开销里大头是镜像 ID 重映射，它随镜像大小线性增长；镜像又是为「任务环境可复现」服务的。这是一场「可复现性 vs 启动速度」的结构性矛盾，靠调参无法解决。

Insight#3 工具时间占比随模型推理速度反向变化（GLM 34.9% 高于 Haiku 19.8%），意味着模型越快，OS 层瓶颈越暴露——「换更强的模型」不仅不掩盖、反而会放大系统问题（笔者推断）。

6.3 Bash 一家独大与工具语义：Agent 的时间表就是一张命令清单

再把「工具执行」这块放大看。Claude Code 提供 Read（读文件）、Edit（改文件）、Grep/Glob（搜索）、Bash（执行任意 shell 命令）、sub-agent（派生子 Agent）等工具。测量结果一边倒：GLM 的 98.8% 工具时间花在 Bash 上；Haiku 则是 Bash 占 80.6% + sub-agent 占 17.3%，两者合计超过 90%。一个 Agent 的时间表，本质上就是一张 Bash 命令清单。

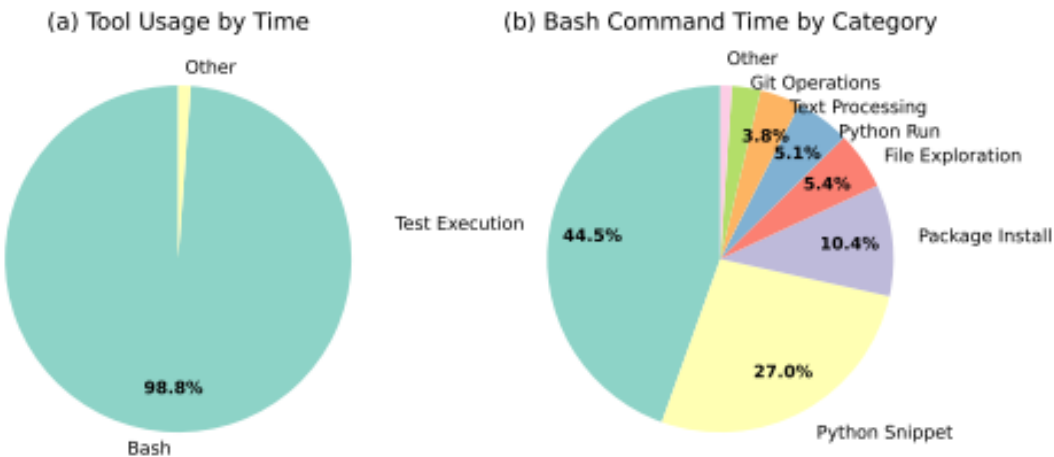


图 2（论文原图）：工具执行时间分布 (a) 与 Bash 命令语义类别占比 (b)，GLM Agent。

怎么读图 2(a) (左侧饼图)： 它把 GLM 全部 111 个任务的工具执行总时间按工具类型切开，Bash 一家占 98.8%，其余所有工具挤在 1.2% 的「Other」里。这说明尽管框架提供了丰富的专用工具，真正吞噬时间的是那个「万能后门」——Bash 可以执行任何命令。

怎么读图 2(b) (右侧)： 它把 Bash 时间再按命令语义分类。测试执行 (pytest、unittest 等) 占 44.5% (Haiku 那边更极端，占 73.3%)，Python 代码片段 (snippet) 占 27.0%，包安装约 10.4%，文件探索 5.4%，Python 运行 5.1%，文本处理 3.8%，Git 操作归在尾部。这张图的潜台词是：Agent 的大部分 OS 开销在做一件事——**反复跑测试验证自己的修改**。

工具之间的执行时长跨越三个数量级：sub-agent 调用平均约 18 秒，Bash 命令 4–6 秒，而 Read、Edit 这类轻量工具不到 0.5 秒。时长差距对应着资源需求差距：跑 pytest 会拉起整个测试进程树、加载项目依赖，读写文件则几乎不消耗什么。这为下一章的「突发式内存」埋下伏笔——时间尺度悬殊的负载混在一起，任何静态资源配额都顾此失彼。

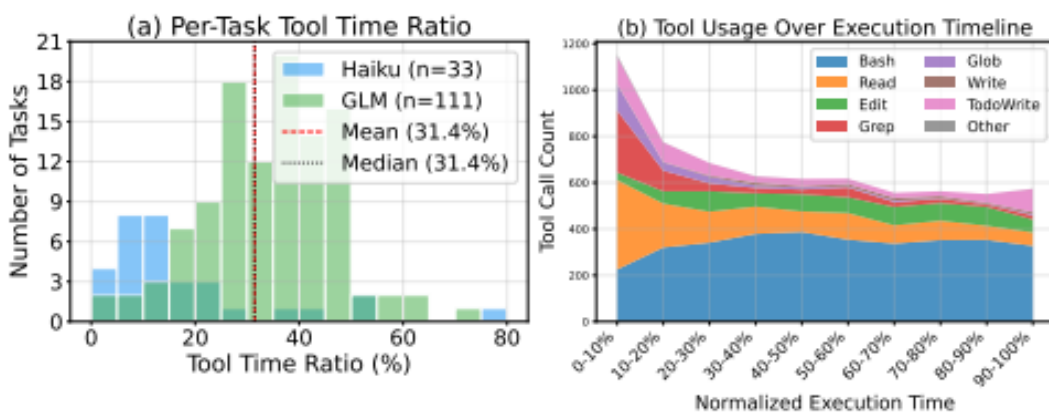


图 3 (论文原图)：逐任务工具时间占比分布 (a) 与执行时间轴上的工具调用分布 (b)，全部 144 个任务。

怎么读图 3(a) (左侧直方图)： 横轴是单个任务的工具时间占比 (0–100% 分桶)，纵轴是任务个数。均值与中位数都是 31.4%，但分布很散——有的任务工具占比几乎为 0% (纯靠 LLM 想)，最高的任务达到 86%。这再次说明任务间的巨大方差：不存在「典型任务」。

怎么读图 3(b) (右侧面积图)： 横轴是把所有任务的执行进度归一化到 0–100%，纵轴是该进度位置上的工具调用次数，不同颜色代表不同工具。可以清楚看到时间模式：Read (读代码) 集中在前 30%——Agent 先「理解」；Bash (跑测试) 在 40–80% 区间达到峰值——中段「验证」；Edit (改代码) 则在整个时间轴上大致均匀分布——「修改」穿插始终。论文称之为「understand-modify-verify」(理解-修改-验证) 模式。

【零基础知识补给站】

什么是 pytest 和单元测试？软件工程里，开发者会为代码写一批自动化检查——每个检查叫一个「单元测试」，输入某个值、断言输出正确。pytest 是 Python 生态最流行的测试框架，一条 pytest 命令就能把项目里成百上千个测试全部跑一遍。SWE-rebench 的任务本质是「修 issue」，而验证修没修对的唯一客观标准就是跑测试，所以 Agent 自然会花接近一半的时间在测试执行上 (GLM 44.5%、Haiku 73.3%)。什么是「理解-修改-验证」模式？这是人类程序员修 bug 的经典三段式，也是 2023 年 ReAct 论文^[38]提出的「推理-行动」(Reasoning + Acting) 交替循环在编程场景的具体化。ReAct (ICLR 2023) 首次系统证明了让 LLM 交替进行推理与外部动作的有效性，本文的图 3(b) 则从系统测量角度证实：真实编程 Agent 确实按这个节奏运行，而且每个阶段的资源画像完

全不同（读文件轻、跑测试重）。最后说说「堆叠面积图」的读法：横轴是时间，纵轴是把此刻各类活动的数量叠起来的总量，每种颜色一层。哪层厚，说明那个时刻哪类活动占主导。读这种图不要看绝对高度，要看颜色层次的相对位置变化——它讲的是「节奏」而不是「数量」。

Insight#1 Bash 占 98.8% 意味着框架精心设计的专用工具（Read/Edit/Grep）在时间上几乎可忽略——资源控制的抓手可以收敛到「Bash 命令」这一个入口，这是后文 bash wrapper 设计的经验依据。

Insight#2 测试执行独占 Bash 时间的 44.5%–73.3%，说明编程 Agent 的 OS 负载本质是「CI（持续集成）负载」——但传统 CI 是批量排队跑的，Agent 却在单个容器里串行反复跑，资源特征完全不同。

Insight#3 「理解-修改-验证」的时间模式意味着资源需求在时间轴上可粗略分段：前 30% 轻、中后段重。这给了资源系统一个「阶段感知」的粗粒度先验，但 § 3.4 将证明这种先验靠不住。

Insight#4 工具时长跨三个数量级（0.5s 到 18s），说明任何「一刀切」的超时或配额设置都会对某一端严重失真——粒度问题在时间上同样存在。

第 7 章 发现二：资源动态——内存才是瓶颈

上一章看的是「时间」，本章对应论文 § 3.3，看的是「资源」：CPU 和内存到底怎么用、怎么涨、怎么落。这是全文数据密度最高的一章，也是 AgentCgroup 所有设计决策的经验地基。结论先行：**限制 Agent 并发密度的是内存而不是 CPU；内存使用呈「185 MB 框架基线 + 工具调用突发」的两层结构；突发由命令语义决定、在 1-2 秒内完成、峰值可达均值的 15.4 倍。**

7.1 内存而非 CPU 决定并发密度

先看 CPU。Agent 的平均 CPU 利用率低得出乎意料：Haiku 13.2%、GLM 7.6%（归一化到一个核，即 100% = 一个核打满）。实验机有 24 个核，也就是说 Agent 大部分时间在「等」——等云 API 返回、等测试进程跑完——而不是在算。CPU 远远没有饱和。

再看内存。单个 Agent 的峰值内存可达 2-4 GB。假设一台 128 GB 内存的机器按峰值做保守分配，最多只能容纳 32-64 个并发实例。而在这个并发度下，CPU 总利用率仍然低于 36%——也就是说**内存先满了，CPU 还闲着三分之二**。对多租户云平台来说，「每个实例能开多少个 Agent」直接决定成本，这个数字被内存锁死，与 CPU 无关。这与传统云负载的经验恰好相反：Azure VM 的研究^[9]（Resource Central, SOSOP 2017，微软对内部虚拟机负载的大规模测量，本文将其作为微服务负载的对照）表明传统负载通常 CPU 与内存都比较均衡，而 Agent 把「内存是唯一硬约束」这件事推到了极端。

【零基础知识补给站】

「并发密度」是什么？想象一个云厂商卖 Agent 算力：一台物理机同时跑越多 Agent 实例，摊到每个实例上的成本就越低。能跑多少个，取决于哪种资源先被用光——那种资源就是「瓶颈资源」。CPU 是「时间资源」：一个核一秒内可以轮流服务很多进程，超卖了大家只是变慢，不会死。内存是「空间资源」：一个字节被占了就是被占了，进程说「我要 4 GB」你就得真的给，给不出就直接 OOM（Out Of Memory，内存耗尽）杀进程。所以内存瓶颈比 CPU 瓶颈「硬」得多：CPU 超卖只是体验下降，内存超卖是任务死亡。为什么 Agent 的 CPU 这么闲？因为它的工作节奏是「问一下模型、跑一个命令、等结果」，大量时间消耗在网络往返和等待子进程上，CPU 真正在算的时间很少。这也是为什么本地 GPU 推理的 GLM（7.6%）比云 API 的 Haiku（13.2%）CPU 还低——云 API 的响应解析和上下文管理反而要吃本地 CPU。对运营者的启示很直接：给 Agent 平台堆更多核没有意义，加内存、管好内存，才是提升密度的唯一杠杆。

Insight#1 「CPU 13% 不到、内存却要 2-4 GB」的组合颠覆了传统容量规划（capacity planning）的经验法则：CPU 与内存配比按 CPU 设计的机型，对 Agent 负载是结构性浪费。

Insight#2 按峰值分配 128 GB 只能跑 32-64 实例，这个数字给所有「Agent 即服务」平台的单机密度划了硬上限——除非资源系统能安全地按均值而非峰值超卖内存，这正是后文 AgentCgroup 的动机。

Insight#3 CPU 利用率低不代表「好管」：低均值掩盖了瞬时峰值超过 175%（多核）的事实（§ 3.3 与 Table 1），只看均值做调度会被突发打穿。

7.2 两层结构：185 MB 框架基线 + 工具突发

论文把 Agent 的内存使用分解为清晰的两层。第一层是**框架基线**：Claude Code 的 Node.js 运行时常驻约 185 MB（Haiku 183 MB、GLM 188 MB，全部 144 个任务早期执行段的平均值），这个值非常稳定，像一条平直的地平线。第二层是**工具突发**：资源波动几乎全部来自工具调用拉起的子进程——跑测试、装依赖等操作会把内存抬到 500 MB–2 GB，命令结束后又落回基线。

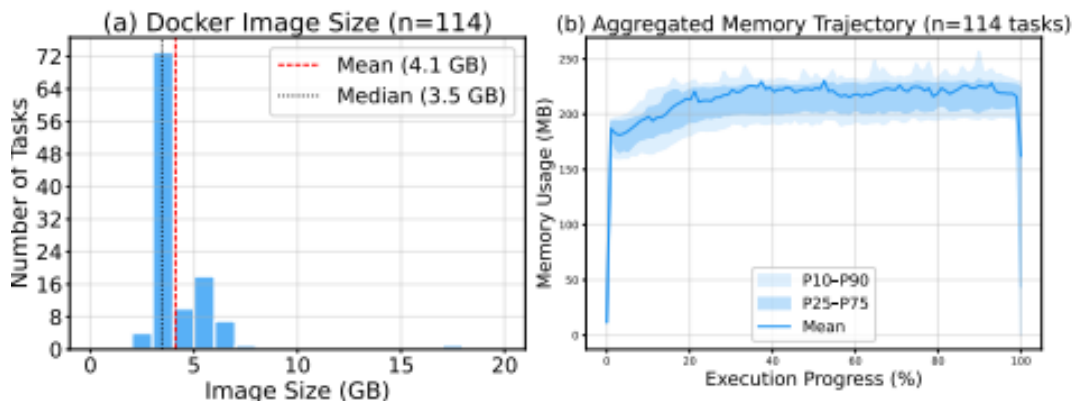


图 4（论文原图）：Docker 镜像大小分布 (a) 与聚合内存轨迹 (b)，全部 144 个任务。

怎么读图 4(a)（左侧）：横轴是镜像大小（GB），纵轴是任务数。Agent 任务镜像集中在 3–4 GB（均值 4.1 GB、中位数 3.5 GB），范围从 2.9 GB 一直到 17.3 GB（114 个去重镜像）。作为对照，典型微服务镜像是 100 MB–1 GB，serverless 函数只有约 50 MB——Agent 镜像是微服务的 7 倍、serverless 的 70 倍。镜像大意味着初始化慢（呼应上一章 31–48%），也意味着「靠多开容器细分粒度」这条路被堵死。

怎么读图 4(b)（右侧）：横轴是归一化执行进度（0–100%），纵轴是内存（MB），图中画出了全部任务的均值线和 P10/P25/P75/P90 分位带。前半程所有分位线都挤在 185–200 MB 附近——这是基线层；后半程高分位带（P90）明显张开——因为 Bash 密集的「验证」阶段开始发力，突发越来越多。注意均值线始终不高：突发是少数时刻的事，但恰恰决定着你必须预留多少内存。

两层结构在多租户场景下的算术很扎眼：64 个并发实例，**光是基线就要占约 12 GB**（185 MB × 64），这还没算任何工具突发；而突发叠加起来的峰值需求比基线高一个数量级。基线是「保底价」，突发是「不可预测的关税」。

【零基础知识补给站】

为什么 Node.js 框架本身就要占 185 MB？Node.js 是一个 JavaScript 运行时，进程启动时就要加载 V8 引擎（Chrome 的 JavaScript 引擎）、即时编译器（JIT）的缓存、框架自身的模块和对话上下文缓冲区。这些内存与「干什么活」无关，只要 Agent 活着就在——所以叫基线。理解「分位带」：把所有任务在同一进度位置的内存值排个序，P50（中位数）是正中间那个，P90 是「90% 的任务都不超过它」的那个值。P90 与 P50 的差距越大，说明「倒霉的那批任务」比「典型任务」重多少——图 4(b) 后半程 P90 张开，就是方差在扩大的可视化。还要理解「overlay 镜像」为什么大：每个任务的镜像里装着一个完整仓库的代码、全部依赖、测试环境，有些还带数据集（所以最大的到 17.3 GB）。这是用空间换可复现性的工程选择，代价就是启动时的 ID 重映射开销和存储压力（111 个 GLM 任务的镜像合计 456 GB）。最后，两层结构对资源管理是个好消息加一个坏消息：好消息是基线稳定、容易规划；坏消息是所有风险都集中在突发层，而突发层恰好是最难管的那

部分——这正是第 9 章「粒度错位」要展开的矛盾。

Insight#1 185 MB 基线 + 工具突发的两层结构意味着「Agent 空闲时的成本」和「干活时的成本」完全解耦——计费与配额体系都应该反映这个二分，而不是用一个混合指标。

Insight#2 64 并发仅基线就 12 GB：基线虽稳，却是多租户密度的「隐形地板」。想把密度翻倍，压基线（比如更瘦的 Agent 框架）和压突发同样重要，而前者目前无人研究（笔者推断）。

Insight#3 镜像平均 3.5 GB 不仅是磁盘问题，它同时抬高了初始化开销（§ 3.2）和堵死了「多开容器」的粒度方案（§ 4.1）——一个大数字引发三处连锁后果。

7.3 决定资源的是「命令做什么」而非「哪个工具」

这一节是 § 3.3 里最反直觉的发现。直觉上我们会按工具类型来预估资源——「Bash 重、Read 轻」。论文证明这是错的：同样是 Bash 调用，仅因执行的命令不同，峰值内存可以相差 13.7 倍。医学影像库 pydicom/pydicom#2022 任务的 Bash 调用平均峰值 4 GB，而网络流工具 streamlink/streamlink#2160 只需 291 MB。工具名是壳，命令语义才是芯。

按命令语义拆开看（呼应图 2(b) 的分类）：测试执行（pytest 等）的 P95 内存尖峰达 518 MB（Haiku）/ 234 MB（GLM），同时伴随平均 +3.2% 的 CPU 尖峰；包安装 P95 尖峰约 233 MB；而文件探索平均只有 4.5 MB，Git 操作 13.5 MB——两者相差两个数量级。换句话说，git status 和 pytest 从「工具视角」看是同一种调用（都是 Bash），从「资源视角」看却是完全不同的物种。

这个发现直接宣判了一类方案的死刑：任何按「工具类型」设置资源配额的系统（「给 Bash 分 512 MB 应该够了吧」）都必然失败——配额必须理解命令的语义，或者干脆放弃静态配额。论文把这个观察称为 tool-semantic variation（工具语义差异），它是第 9 章 Table 2 中「粒度错位」那一行的关键论据。

【零基础知识补给站】

什么是「P95」？把所有测量值从小到大排序，排在 95% 位置的那个值就是 P95——它回答的是「绝大多数（95%）情况下不超过多少」。系统工程师喜欢 P95/P99 胜过最大值，因为最大值可能被一次极端异常带偏，P95 更代表「需要认真应付的坏情况」。这里 pytest 的 P95 = 518 MB 意味着：如果你按 518 MB 给测试调用预留内存，100 次里大约 95 次够用，但还有 5 次会爆。为什么跑测试这么吃内存？pytest 会导入整个项目及其依赖树，把测试数据加载进内存，有时还会并行拉起多个进程。科学计算类项目（numba、pydicom）的测试会构造大数组、读医学影像，内存轻松上 GB；而一个命令行小工具（faker）的测试只需要造点假数据，百 MB 以内搞定。「语义决定资源」这个原则其实在生活中随处可见：同样是「开车」，去楼下超市和拉一车水泥上高速，油耗天差地别——你不能因为都是「开车」就给所有车配一样大的油箱。论文的贡献是把这件「常识」变成了精确的定量结论（13.7×、518 vs 4.5 MB），并指出它破坏的对象：所有基于工具类别做资源预留的方案。

Insight#1 「工具类型」是资源需求的坏代理变量，「命令语义」才是好代理变量——但语义在 LLM 生成命令的那一刻才确定，这决定了静态配额在原理上不可行。

Insight#2 pydicom 4 GB 与 streamlink 291 MB 跑在同一规格的容器里，说明 SWE 类基准的「环境统一性」掩盖了需求的极端异质性——环境是平的，需求是尖的。

Insight#3 Git 13.5 MB 与 pytest 518 MB 的 40 倍差距提示：如果系统能在命令启动前识别语义类别，就已经能拿下大部分预测收益——这正是 AgentCgroup 意图声明（hint）机制的用武之地。

7.4 burst-silence 模式：98.5% 的内存突发发生在工具调用内

把内存的时间序列与工具调用的时间轴对齐，论文发现了高度的时空对齐：**内存尖峰几乎只在工具调用期间出现，LLM 推理阶段则平稳而低**。定量地说（以超过 300 MB——约 1.6 倍框架基线——定义一次内存突发）：Haiku 任务里工具调用只占 28.5% 的采样时间，却包含了 98.5% 的内存突发；GLM 任务里 35.8% 的时间包含 67.3% 的突发（1.9 倍集中度）。CPU 突发的归因更分散（Haiku 55.3%、GLM 30.2%），因为 GLM 的本地推理在工具调用之外也产生持续的 CPU 负载。单个 1 秒采样间隔内，内存变化可达 2.9 GB，CPU 峰值可超过 100%（多核）。

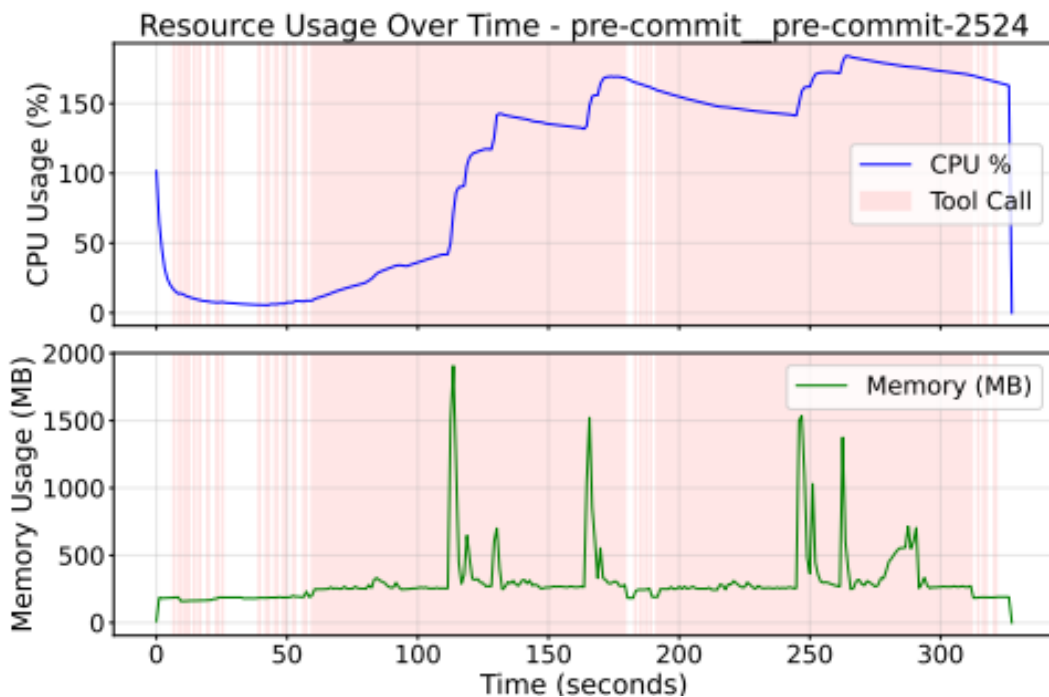


图 5（论文原图）：资源使用时序——Haiku Agent 执行 pre-commit/pre-commit#2524。

怎么读图 5：上下两个子图共享同一根时间轴（秒）。上图是 CPU 利用率（%），下图是内存（MB）；粉色（浅色）竖带标记的是工具调用发生的时段。先看内存图：平段约 200 MB 上下——那是框架基线；几座拔地而起的「尖塔」无一例外都落在粉色竖带内，最高的接近 2000 MB，离开粉带立刻回落。再看 CPU 图：尖峰同样集中在粉带里，但也有零星粉带外的波动。整张图是「burst-silence」（突发-静默交替）模式最直观的肖像：任务的生命就是「安静—被工具点燃—回落—再安静」的循环。

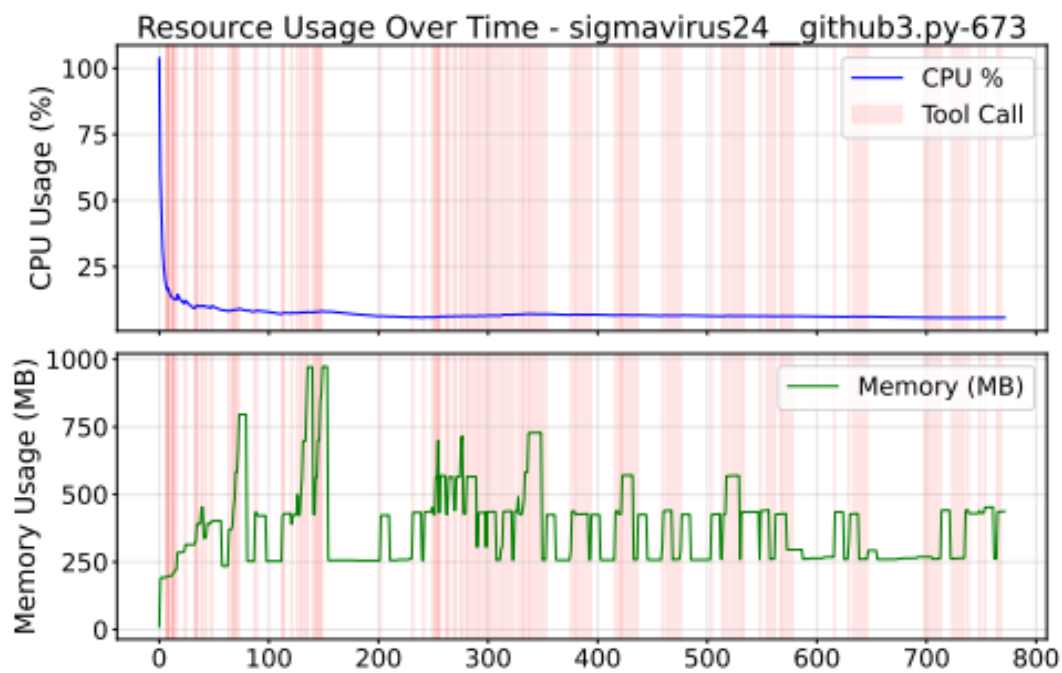


图 6（论文原图）：资源使用时序——GLM Agent 执行 sigmavirus24/github3.py#673。

怎么读图 6（与图 5 对照着读）：这是 GLM 本地模型跑另一个任务（时长约 15 分钟，明显更长）。同样的规律：内存尖峰（这里约到 1000 MB）对齐粉色工具调用区。不同之处在于底噪——GLM 的 CPU 曲线在粉带之间也有持续的小幅起伏，那是本地 GPU 推理伴随的 CPU 辅助开销；而且 GLM 的工具调用更密、时间更长，粉色区域占比更高（35.8% vs 28.5%）。两张图合起来传递的信息是：突发-静默模式是跨模型成立的，但突发的「浓度」和 CPU 底噪随模型部署方式而变。

这个模式的价值在于「可定位」：既然 98.5% 的内存突发都被圈在工具调用的时间窗里，那么把控制点设在工具调用边界上，就几乎不会漏掉任何一次资源事件。这是 AgentCgroup 选择「以工具调用为控制域」的直接经验依据——负载天然地把边界画好了。

【零基础知识补给站】

什么是「burst-silence」模式？把任务想象成一间办公室：大部分时间大家都在安静办公（LLM 思考，内存稳定在 185 MB），偶尔搬进来一队装修工人（工具调用，内存冲上 GB 级），干完活立刻撤走。这就是「突发-静默」交替。与之相对的典型云负载是「steady」（稳态）：一个 Web 服务的内存常年在同一水平线上小幅波动。稳态负载好管——按平均值加一点余量就行；突发负载难管——按均值配会在突发时爆掉，按峰值配则 98% 的时间在浪费。什么是「采样」？论文每秒记录一次 CPU 和内存数值（1 Hz 采样），然后给每个采样点贴标签：此刻属于 LLM 推理还是工具调用。这样就可以统计「突发落在哪类时段里」。需要注意 1 秒采样会漏掉更短的突发——实际波动只会比观测到的更剧烈，所以论文的数字是保守的下限。最后，「98.5% 的突发在工具内」还有一个隐含好处：工具调用对框架来说是「已知事件」（框架自己发起的），不需要系统去猜——把资源钩子挂在工具边界上，工程上完全可行，这为第 10 章的 bash wrapper 方案提供了正当性。

Insight#1 突发-静默模式 + 98.5% 的工具内归因，意味着资源控制的最好附着点不是时间片（定时巡检）而是事件（工具调用边界）——控制论上的「事件驱动」完胜「轮询」。

Insight#2 GLM 突发集中度只有 67.3%（vs Haiku 98.5%），说明模型部署方式会「稀释」这一模式的纯度：本地推理把一部分负载挪出了工具窗口。控制方案若过度依赖边界假设，换模型就要打折（笔者推断）。

Insight#3 CPU 突发归因分散（Haiku 55.3%、GLM 30.2%）提醒我们：内存和 CPU 不能用同一套边界假设管理——这为 7.6 的相关性分析做了铺垫。

7.5 突发的速度与幅度：1-2 秒、15.4×、3 GB/s

突发有多猛？三个数字足以让人倒吸凉气。第一，**时长只有 1-2 秒**——一次内存尖峰从升起回到落，比人眨几次眼还快。第二，**峰均比（peak-to-average ratio）最高 15.4×**：最极端的 pydicom/pydicom#2022 峰值内存 4060 MB，而任务平均内存只有 264 MB；峰值过后几秒内就回落到 230 MB 基线。第三，**变化率最高 3 GB/s**——一秒内内存占用可以垂直拉升 3 个 GB；CPU 变化率超过 50%/秒；有 1.7%-3.8% 的 1 秒采样间隔内存变化超过 100 MB。内存峰值集中在执行后半程（均值/中位数约在 65% 进度处），但整个执行周期都可能出现。

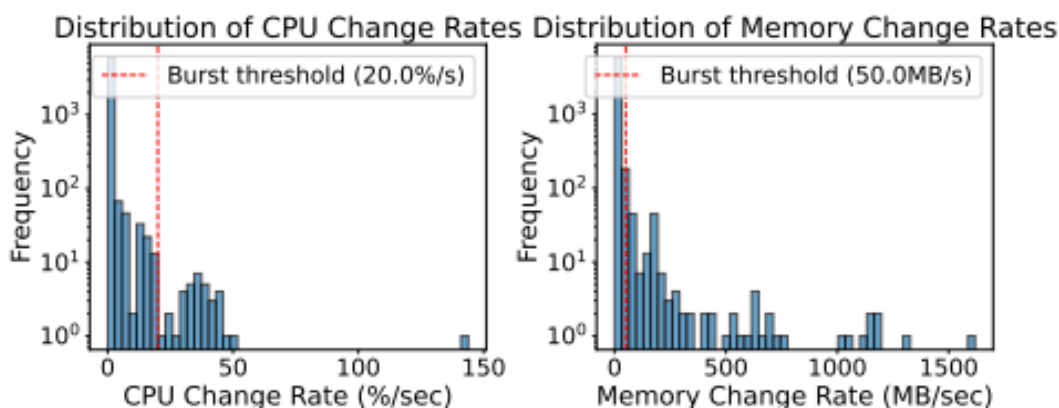


图 7（论文原图）：资源变化率分布（CPU 与内存），Haiku 数据集。

怎么读图 7：上下两张都是频率直方图，横轴分别是「每秒 CPU 变化率（%/s）」与「每秒内存变化率（MB/s）」，纵轴是对数刻度的频次。注意横轴一直延伸到 150%/s 和 1500 MB/s 以上——分布拖着长长的右尾。图中的虚线是论文定义的突发阈值线（CPU 20%/s、内存 50 MB/s）：虚线以右的长尾就是「突发区间」。尾部虽不粗，却实实在在存在——绝大多数采样点风平浪静，但那 1.7%-3.8% 的间隔足以杀死任何反应慢半拍的控制器。

把这三个数字放到一起，就能体会到传统控制器的绝望：一个 1-2 秒完成、速度达 3 GB/s 的突发，等用户态监控程序发现异常、开会讨论、下发指令，突发早就结束或者已经把 OOM killer（内存耗尽杀手）招来了。这组数字是第 9 章「响应速度错位」的全部论据，也是第 10 章选择「内核态 eBPF 微秒级反应」的全部理由。

【零基础知识补给站】

什么是峰均比（peak-to-average ratio）？峰值除以均值。它衡量负载的「尖刺程度」：峰均比 1× 是完全平稳，越大越尖。作为参照系：Azure 无服务器函数的内存峰均比约 1.5×^[30]（前文提到的微软 serverless 测量，几乎平的），Azure 虚拟机 2-3×^[9]，Google Autopilot 的推荐余量也

在实际峰值的 $2\times$ 以内^[29] (Autopilot 是 Google 2020 年 EuroSys 论文提出的集群自动调容系统，本文用它作为「历史预测法」的代表靶子)。而 Agent 是 $15.4\times$ ——比传统云负载高出一个数量级，超出了所有现有系统的假设范围。为什么内存回收跟不上 3 GB/s? Linux 回收内存 (reclaim) 要遍历页链表、判断哪些页面可以丢弃或写回磁盘 (swap)，这是逐页处理的细活；进程分配内存却只需要在页表里登记。分配是 O (登记)，回收是 O (扫描)——两者的速率天生不对称。当分配速率达到 GB/s 量级，回收线程还没热身，内存就已经见底，OOM killer 直接动刀。这就是为什么「靠回收兜底」对 Agent 突发不成立。最后注意 1-2 秒这个时间尺度与人、与监控系统的对比：人类的反应时间约 0.2 秒，Prometheus 这类监控的默认抓取间隔是 15-30 秒，Kubernetes 的自动扩缩容决策周期是分钟级——全都慢了一个到四个数量级。

Insight#1 $15.4\times$ 的峰均比把 Agent 负载推出了所有已有容量规划模型的假设区间 ($\leq 2-3\times$)，这不是「更差的参数」而是「失效的模型」——修补旧模型不如换模型。

Insight#2 3 GB/s 的分配速率对 1 Hz 采样意味着：监控体系看到的永远是突发的「尸体」而非「现场」——观测基础设施本身就需要为这种负载重新设计 (更高频、更内核态)。

Insight#3 峰值集中在 65% 进度附近但又「全程皆有可能」，说明「后半程多加内存」这种粗略启发式只能降低概率、不能消除风险——不确定性是结构性的，不是噪声。

7.6 重试循环与内存爬升：时间会发酵，内存会积食

最后一个动态特征来自 Agent 的「倔脾气」：测试没过就再跑一遍。论文定义「重试组」 (retry group) 为连续三次以上执行同一条测试命令的 Bash 调用 (例如反复跑 pytest)。统计结果：85% (28/33) 的 Haiku 任务、97% (108/111) 的 GLM 任务包含重试组；GLM 平均每任务 3.9 个重试组，最多出现 56 次连续重试；重试消耗总执行时间的 7-21% (Haiku 7.4%、GLM 20.5%)。

更要命的是：每一轮重试都保留着上一轮的内存上下文而不清理，造成渐进式内存爬升 (progressive memory accumulation)，最坏情况下有 502 MB 内存始终未释放。这意味着「第 5 次重试时的内存需求」高于「第 1 次时」——一个在任务早期够用的内存上限，可能在第 5 次重试时触发 OOM。内存需求在单次执行内部也是漂移的。

本节最后一个发现拆掉了「CPU 与内存同涨同落」的隐含假设：逐任务计算 CPU 与内存的相关系数，范围从 -0.84 到 +0.50，平均 -0.39。有些任务正相关 (工具执行同时拉高两者)，另一些任务负相关 (CPU 密集阶段恰好内存需求低)。**不能用 CPU 的忙碌程度推断内存压力，反之亦然**——两种资源必须独立测量、独立控制。这也是为什么把 CPU 和内存打包成一个「实例规格」 (如 Kubernetes QoS 的做法，见第 9 章) 对 Agent 负载格外不合适。

【零基础知识补给站】

为什么内存不像 CPU 可以「超卖」？这是本章反复出现的潜台词，值得展开。CPU 是时间资源：一秒钟可以切成一千片轮流分给不同进程，超卖的结果是大家排队变慢，但没人死。内存是空间资源：进程申请到的一页内存是被「独占」的，系统不能趁它不注意借给别人用一下再还回来。所以内存超卖没有「温柔降级」的选项，只有 swap (把内存页换到磁盘，慢一百倍) 和 OOM (直接杀进程) 两条路。Agent 的突发又快又猛，swap 根本来不及。什么是相关性系数 r ？它衡量两个变量「一起动」的程度： $r = +1$ 是完全同涨同落， $r = -1$ 是完全此消彼长， $r = 0$ 是互不相关。论文实测的范围 -0.84 到 +0.50 说明：不同任务里 CPU 与内存的关系可以是任何形状——甚至同一个平台上都

找不出统计规律。「内存爬升」又是什么？程序跑完一轮不把所有内存还给系统（可能是缓存、可能是内存泄漏、可能是语言运行时故意留着的堆），下一轮重试在已经膨胀的堆上继续加。就像反复往桌上堆文件却不收拾，第 56 次重试时桌子早就放不下了。对运维的启示：监控 Agent 任务时，「内存随时间缓慢单调上升」不是异常而是常态；传统的「内存增长率告警」阈值需要按任务阶段校准，否则全是误报（笔者推断）。

Insight#1 97%（GLM）的任务含重试组，说明「反复试错」不是边缘行为而是 Agent 的工作方式本身——资源系统必须为「同一条命令的第 N 次执行」预留递增的预算，这与所有无状态假设相悖。

Insight#2 56 次连续重试 + 502 MB 未释放内存揭示了一个尴尬事实：Agent 不知道自己陷入了循环，系统也没有机制告诉它——「让 Agent 感知资源」不仅是优化，还是止损（笔者推断）。

Insight#3 CPU-内存相关性均值 -0.39 且跨正负号，意味着任何把两种资源捆绑定价/捆绑配额的产品形态（实例规格、QoS 档位）对 Agent 都是错配——解耦控制是必要条件而非可选项。

Insight#4 重试占 7-21% 的执行时间是一笔「隐形学费」：它既推高时延又推高内存，却几乎不产生新信息。降低重试率可能是 Agent 工程里被低估的性价比优化点（笔者推断）。

第 8 章 发现三：不可预测性——历史在这里不作数

前两章回答了「Agent 负载长什么样」（时间构成、资源动态），本章对应论文 § 3.4，回答一个更尖锐的问题：**它的资源需求能不能被预测**？答案是斩钉截铁的「不能」。这个「不能」不是工程上的暂时困难，而是从同一任务的重复运行、可观测代理指标、跨任务方差三个层面被反复证实的结构性结论。它直接宣判了「基于历史的资源预测」——过去二十年云资源管理的主流范式——在 Agent 负载上的死刑。

8.1 同一任务三次运行：402、222、259 秒，三条完全不同的路

论文做了一个简单却杀伤力十足的实验：把同一个任务（iterative/dvc#777）原封不动跑三遍。结果执行时间分别是 402 秒、222 秒、259 秒——最大相差 1.8×。如果只是快慢不同倒也罢了，更颠覆的是**三次运行产生了完全不同的解法**：修改的代码不同、改动的文件数不同、整体策略不同。同一个 Agent、同一份任务描述、同一个起点，走出了三条不相交的路径。

论文把原因归结为两点：LLM 推理的随机性（sampling randomness）与决策路径的多样性。模型每生成一个词都是一次带随机性的抽样，早期的微小差异会被后续的「推理-行动」循环不断放大——第一次跑选择了「先修 A 文件再跑测试」，第二次可能选择「先写个复现脚本」，之后的一切就完全不同。这是一种混沌式的敏感依赖：**Agent 任务的资源需求在执行开始之前，原则上就是未知的**。

【零基础知识补给站】

LLM 的随机性从哪里来？LLM 生成文本时，每一步都要从几个候选词里挑一个。它不是总挑概率最高的那个，而是按概率分布「抽签」——抽签的激进程度由一个参数 temperature（温度）控制：温度低时几乎总选最高概率的词（接近确定），温度高时更敢选冷门词（更有创造性也更不稳定）。即使温度设为 0，工程实现中的浮点误差、并发请求混合等因素也会引入残余不确定性。对聊天机器人，这点随机性只是「换种说法」；对编程 Agent，一次不同的措辞可能意味着选了不同的命令、读了不同的文件，然后滚雪球：不同的命令 → 不同的输出 → 不同的后续推理 → 完全不同的解法。这就是为什么 dvc#777 三次运行能差出 1.8 倍时长。对比一下传统程序：你跑一万遍「排序 100 万个数」，每次的指令序列、内存曲线几乎一模一样，这就是「确定性负载」——云厂商可以安全地用昨天的测量规划明天的容量。而 Agent 把「昨天」变得毫无意义：连重放同一个任务都复现不了自己。理解这一点，才能理解为什么论文把「不可预测性」单独列为一大发现，并据此推翻了第 9 章的「适应性错位」。

Insight#1 1.8× 的同任务方差是一个下限：它只测量了「同一任务同一框架」的重复性，尚未计入换模型、换提示词带来的更大方差——真实生产中的不确定性只会更高。

Insight#2 三次运行产出三种解法，说明 Agent 的「任务」概念本身就比传统负载弱：传统负载里「任务」定义了工作，Agent 里「任务」只定义了目标，路径是即兴的。

Insight#3 对资源系统的推论很残酷：不仅「这个任务要多少内存」不可知，连「这次运行」与「上次运行」是否可比都成问题——预测失去了最基本的统计前提（独立同分布）。

8.2 可观测代理指标全部失效：看得见的不顶用

也许有人会说：资源需求预测不了，那就找一个「代理指标」（proxy）——某个容易观测、又和资源需求强相关的量。云平台一直这么干，例如用「请求数」预测负载。论文系统检验了 Agent 场景下最自然的两个候选，结论是全部失效。

第一个候选是**对话轮数**（conversation rounds）。它与执行时间的相关性中等偏强： $r = +0.57$ 到 $+0.82$ ——聊得越久确实跑得越久。但它与峰值内存的相关性 $r < 0.11$ ，几乎为零。也就是说「这个 Agent 想了多少轮」完全不能告诉你「它需要多少内存」。第二个候选是**输出 token 数**：它与峰值内存的相关系数 $r = -0.14$ ，同样约等于零。

为什么失效？论文的解释一针见血：**资源消耗由「工具执行了什么」驱动（pytest 还是读文件），而不是由「推理的规模」驱动**。一轮只输出 20 个 token 的对话可能触发一次 4 GB 的 pytest；一段滔滔不绝的长推理可能之后只接了一个 git status。可观测的是「想」，决定资源的是「做」，两者之间没有稳定的统计桥梁。

【零基础知识补给站】

什么是「代理指标」？当你想测的东西太贵、太慢或根本测不到时，找一个便宜又好测、且（hopefully）与它高度相关的指标代替。比如医院用「体温」代理「感染」，云服务用「在线用户数」代理「服务器压力」。代理指标成立的前提是相关性要稳定——如果体温和感染脱钩了，体温计就废了。论文用相关系数 r 检验了两个候选：对话轮数和输出 token 数。 $r = +0.57 \sim +0.82$ 算「中等偏强的正相关」——轮数多确实时间长，这在直觉上成立；但 $r < 0.11$ 和 $r = -0.14$ 就是「统计上等于没关系」。一个能预测时间却不能预测内存的指标，在内存才是瓶颈（第 7 章）的世界里价值有限。这个结论的杀伤力在于：Agent 框架自己能拿到的全部信息（轮数、token、工具次数）都在「思考侧」，而资源压力在「行动侧」。框架看得到的，预测不了资源；决定资源的（命令语义），框架在生成之前也不完全知道——信息不对称是双向的。这正是第 10 章「双向协议」设计的动机之一：既然单向的观测推断走不通，就让系统把资源压力「告诉」Agent，让 Agent 把意图「告诉」系统。

Insight#1 「轮数预测时间、不能预测内存」的分裂说明：Agent 负载里时间与空间是解耦的两个维度——这与第 7 章 CPU-内存弱相关（均值 -0.39 ）互为印证，资源画像需要多维独立建模。

Insight#2 代理指标失效的根源是「思考-行动」的因果不对称：LLM 在生成工具调用之前不知道自己会生成什么，任何基于推理侧特征的预测都是在预测一个尚未做出的决定。

Insight#3 对监控产品的实际影响：市面上按 token 或轮数计费/限流的方案，与真实资源成本几乎没有关系——计费模型与成本模型的错配会成为平台亏损或用户怨言的来源（笔者推断）。

8.3 跨任务 20×、跨模型再放大：方差之上还有方差

把视野从单任务拉到整个数据集，方差继续膨胀。峰值内存需求从 197 MB 到 4 GB，**跨任务相差约 20 倍**，变异系数 CV（coefficient of variation，变异系数）高达 147%。科学计算任务（numba/numba#5721、pydicom/pydicom#2022）的内存显著高于命令行工具（joke2k/faker#1520）或网络工具（streamlink/streamlink#2160）——但它们跑在完全相同的容器规格里，从环境上完全看不出差别。

模型选择则把方差再放大一层。同样的 33 个任务，Haiku 与 GLM 的 CPU 利用率相差 $1.7\times$ （13.2% vs 7.6%）。更深层的差异在 CPU 画像：Haiku 走云 API，本地 CPU 要承担响应解析与上下文管理，有 8.2% 的采样点 CPU 超过 50%；GLM 本地 GPU 推理，CPU 几乎全花在工具调用上，只有 0.5% 的采样点超过 50%。换句话说，**换一个模型，等于换了一种负载**——不仅数值变了，连资源的「形状」都变了。

三层方差叠加（同任务 $1.8\times$ 、跨任务 $20\times$ 、跨模型再放大），让「历史百分位」统计彻底失效。这正是 Google Autopilot^[29]（EuroSys 2020，Google 的集群自动调容系统，用历史 P95 自动推荐资源请求值，本文将其作为预测式方案的对照）这类系统的前提被击穿的地方：Autopilot 假设负载可重复、历史有效，而 Agent 负载三条全都不满足——第 9 章的「适应性错位」将正式展开这场对决。

【零基础知识补给站】

什么是变异系数 CV？标准差除以均值，用百分比表示。它消除量纲，让「波动大不大」可以跨场景比较：CV = 30% 算温和波动，CV = 147% 意味着标准差是均值的 1.47 倍——分布已经散到「均值」这个数字本身失去代表性。作为直觉对照：人类成年男性身高的 CV 大约 4–5%。Agent 峰值内存的 147% 相当于「人群里既有 20 厘米的也有 40 米的」（笔者类比）。为什么「历史百分位」统计会失效？统计学里用「过去的 P95」预测「未来的需求」，有一个不言自明的前提：样本来自同一个分布（独立同分布，i.i.d.）。Agent 负载三重违反这个前提：同一任务重复运行的分布都不稳（ $1.8\times$ ），不同任务根本是不同分布（ $20\times$ 、CV 147%），换模型连分布的形状都换（CPU 画像 8.2% vs 0.5%）。历史数据不是「少」，而是「不属于同一个总体」——再多历史也救不回来。还要理解「跨模型差异」的工程含义：一个为 Haiku 画像调好的资源系统，换成 GLM 后 CPU 假设全部作废，反之亦然。而 Agent 平台恰恰需要频繁换模型（升级、降级、A/B 测试、成本优化），每次换模型都等于重新标定整个资源体系——除非资源系统本身不依赖任何模型特定的先验。这又一次指向第 10 章的「意图驱动」：不猜，问。

Insight#1 CV = 147% 是一个「模型终结者」数字：它意味着任何单点估计（均值、中位数）作为配额依据都会同时造成大面积浪费与大面积超限——分布的宽度已经超过了配额系统的调节能力。

Insight#2 跨模型 $1.7\times$ 的 CPU 差异看似不大，但配合 CPU 画像的形态变化（8.2% vs 0.5% 高负载点）说明：模型选择是负载的一阶决定因素——资源系统必须把「当前是哪个模型」当作头等输入，或者干脆对模型不可知。

Insight#3 三层方差叠加的结构提示了正确的应对顺序：跨任务方差（ $20\times$ ）最大，应靠意图声明解决；同任务方差（ $1.8\times$ ）次之，只能靠实时内核态反应吸收；跨模型方差靠「不依赖先验」的架构免疫——这正是第 10 章设计的三层分工（笔者推断）。

Insight#4 「同一容器规格跑 197 MB 到 4 GB」暴露了基准测试的设计盲区：环境复现性被放在了需求异质性之上——未来的 Agent 基准或许应该把「资源需求」也纳入任务元数据（笔者推断）。

第 9 章 三大错位：为什么现有资源管理全都不适用

第 6–8 章完成了「画像」，本章对应论文 § 4，完成「对决」：把 Agent 负载与现有的每一类资源管理方案逐一对比，论证它们为什么全部不适用。这是全文的论证核心——没有这一章，AgentCgroup 就只是一个「可以这么做」的系统；有了这一章，它变成「只能这么做」的系统。论文把不适用归结为三个结构性错位（mismatch）：**粒度错位（granularity）**、**响应速度错位（responsiveness）**、**适应性错位（adaptability）**。

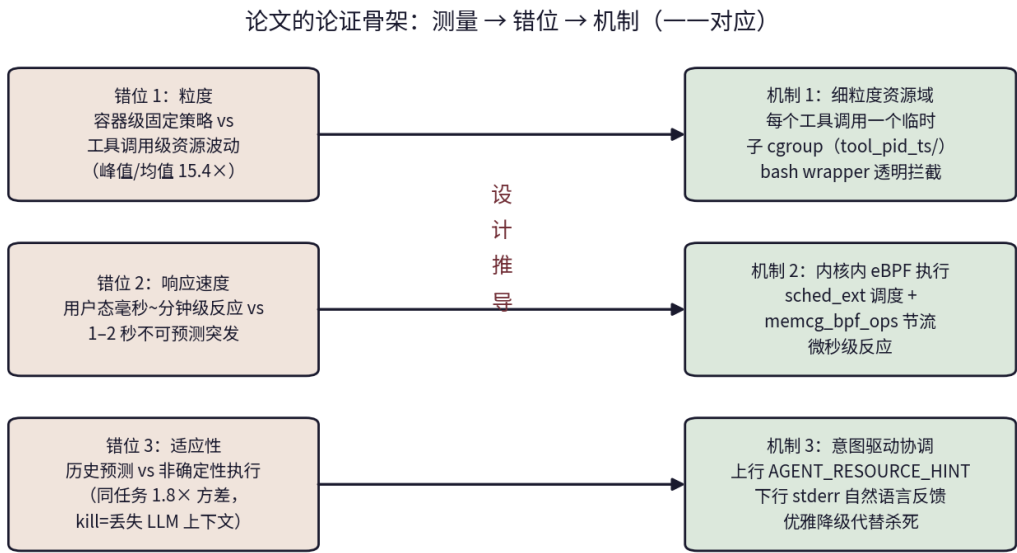


示意图 D4：三大错位总览——Agent 负载的三个特征与三类现有方案的三对结构性矛盾。

怎么读这张图：可以把它当作本章的地图。左侧是 Agent 负载的三个已被 § 3 证实的特征（工具调用级动态、1–2 秒突发、不可预测的有状态执行）；右侧是现有方案的三个家族（静态限额、反应式控制、预测式扩缩容）；中间的连线就是三对错位的对应关系。接下来三节我们沿着这三条线逐条论证。

9.1 与四类传统负载的定量对比：一张表看懂「新物种」

论文把 AI 编程 Agent 与三类经典云负载放在同一张表里，沿八个维度定量对比（Table 1）。表中三个对照组各有出处：serverless 数据来自微软对 Azure Functions 的大规模测量^[30]；微服务数据来自微软的 Resource Central 研究^[9]；批处理数据来自 Google Borg 集群管理论文^[32]（EuroSys 2015，Google 官方披露的集群调度系统，本文用它作为「长时间确定性作业」的代表）。我们把这张表完整重建如下。

维度	Serverless/FaaS [30]	微服务 [9]	批处理/HPC [32]	AI 编程 Agent
执行时长	100 ms–2 s	长期运行	数分钟–数小时	5–11 分钟
容器镜像	约 50 MB	100 MB–1 GB	1–10 GB	2.9–17.3 GB（中位 3.5 GB）
状态性	无状态	状态在外部	有状态	进程内有状态

维度	Serverless/FaaS [30]	微服务 [9]	批处理/HPC [32]	AI 编程 Agent
内存占用	128–512 MB	稳定在约 1 GB	随数据规模伸缩	空闲 185 MB，峰值 2–4 GB
内存峰均比	约 1.5×	2–3×	约 1×	15.4×
CPU 利用率	瞬时尖峰	10–40%	80–100%	均值 <13%，峰值 >175%
确定性	确定	基本确定	确定	同任务 1.8× 方差
资源模式	平坦	稳态 + 昼夜周期	稳定爬升	突发-静默交替
终止代价	重试即可	可迁移	丢失进度	丢失全部 LLM 上下文

表 1（重建自论文 Table 1）：AI 编程 Agent 与典型云负载的定量对比。

逐行看。执行时长：Agent 卡在 serverless 与批处理之间的无人区（第 6 章已展开）。容器镜像：3.5 GB 中位数，是微服务的约 7 倍、serverless 的约 70 倍——大镜像同时推高初始化开销、堵死细粒度拆分。状态性这一行最关键：serverless 无状态（Stateless）、微服务把状态放在外部存储、批处理有状态但可从头重算，而 Agent 是「**进程内有状态**」（in-process stateful）——对话上下文、推理历史都存在进程内存里，杀进程等于销毁它的「记忆」。内存峰均比：1.5× / 2–3× / 1× 对 15.4×，直接超出现有系统的假设区间一个数量级。CPU 利用率：均值 <13% 但峰值 >175%（多核），「均值低、峰值高」的组合让任何单一数字都失真。确定性与资源模式两行互为因果：正因为是「突发-静默交替」且「同任务 1.8× 方差」，确定性列才与其他三类形成鲜明对照。最后一行终止代价（termination cost）是全表的灵魂：serverless 函数被杀了重试就行，微服务实例可以迁移，批处理被杀损失进度，而 Agent 被杀**丢失全部 LLM 上下文且重跑不保证收敛**——终止从「小事故」变成「任务失败」。

【零基础知识补给站】

这张表为什么重要？云计算二十年的资源管理技术，几乎全部是为表中前三列的负载量身定做的：为 serverless 做的预热池与毫秒级冷启动优化，为微服务做的弹性扩缩容与服务网格，为批处理做的排队调度与回填（backfill）。每一代技术都隐含了对所在列负载特征的假设。这张表用八个维度证明：Agent 在任何一列都不属于任何一类——它是新物种，不是「某个旧类别的边缘 case」。几个行内术语解释一下。「无状态」指每次请求独立处理、实例间可互换，所以 serverless 平台可以随意创建销毁实例；「状态在外部」指微服务把数据存在数据库里，进程本身可替换；「进程内有状态」则是状态绑死在活着的进程上——Agent 的 LLM 对话上下文包括几十轮的思考记录和工具输出，只存在于进程内存与 API 会话中，没法导出、没法迁移、没法检查点。「昼夜周期」指微服务流量随用户作息起伏（白天高夜里低），可预测性好；「稳定爬升」指批处理随着数据读入内存逐步增长，单调可外推。理解这些对照后你会发现：表中每一行都是一个「假设被违反」的清单，而第 9 章余下部分就是逐行清算这些违反的后果。

Insight#1 八维度对比的真正结论不是「Agent 更极端」，而是「Agent 的组合不存在先例」：单项看每个指标都有负载接近，但「长时间 + 大镜像 + 进程内有状态 + 高峰均比」的联合分布在云历史上从未出现。

Insight#2 「终止代价」一行是整张表中被低估的战略行：它决定了容错机制的设计空间——前三类负载的容错都建立在「杀得起」上，Agent 的一切资源管理必须建立在「杀不起」上。

Insight#3 把镜像 3.5 GB 与时长 5-11 分钟两行连起来看，得到一个隐藏推论：初始化成本（31-48%）与任务时长的比值天然偏高，任何「多开容器换隔离」的方案都在数学上先输一半。

9.2 错位一：粒度——容器级政策 vs 工具调用级动态

Linux cgroup (control group, 控制组) 是内核的资源治理抽象^[14]：内核把进程组织成树状的控制组，沿层级做记账与限额。内存控制器提供两个关键边界：memory.high 是软限流点（触发回收压力但不调用 OOM killer），memory.max 是硬上限（超限即触发 OOM）。问题在于：**所有这些接口的施力粒度都是「容器级」——一棵树一个政策，而 Agent 的需求在「工具调用级」变动**（第 7 章：Bash 调用间峰值内存差 13.7×）。

先看硬上限 memory.max 的两难。按峰值设（比如 4 GB）：峰值出现的时间占比不到 2%，意味着超过 90% 的已分配内存存在绝大多数时间闲置浪费。按均值设（比如 264 MB）：第一次 pytest 突发就触发 OOM，积累的 Agent 状态全部销毁。重试循环让这道题更无解：渐进式内存爬升意味着「对前几次迭代够用的限额，可能在第 5 次重试时引爆 OOM」（§ 3.3）。

软限 memory.high 也好不到哪去。触发回收时，内核无法区分「稳定的框架基线」（Node.js 堆、V8 JIT 缓存）和「工具子进程的内存」，回收压力会平等地砸向 Agent 运行时本身，造成 GC（垃圾回收）抖动。更根本的是：一个容器级阈值无法区分 git status (13.5 MB) 和 pytest (P95 518 MB)——它俩在同一个容器里，共享同一个数字。

上升到编排层，Kubernetes 的 QoS (Quality of Service, 服务质量) 等级^[19]把 Pod 分为 Guaranteed、Burstable、BestEffort 三档 (K8s 官方文档定义的 Pod 服务质量分级，本文用它代表「静态分级」方案)。Guaranteed 等于按峰值预留，浪费同上；BestEffort 让有状态的 Agent 随时可能被驱逐；Burstable 折中但仍然设不了「按工具调用」的配额，并且三档都把 CPU 和内存绑进同一个等级——而第 7 章已证明两者相关性又弱又变 (-0.84 到 +0.50)。最后，「那多开几个容器把粒度变细行不行？」也不行：镜像平均 3.5 GB、初始化占 31-48%，按工具开容器等于让初始化开销吃掉全部任务时间。结论：**粒度必须细化到容器内部，需要「子容器资源域」(sub-container resource domains)。**

【零基础知识补给站】

什么是 Kubernetes QoS 三档？Kubernetes (K8s) 是最主流的容器编排系统。它根据你为 Pod 设置的资源「请求值 (request)」和「上限值 (limit)」把 Pod 分三档：Guaranteed (请求=上限，最优先被杀保护，适合关键服务)、Burstable (请求<上限，可弹性超出但超卖时先被杀)、BestEffort (不设任何值，资源紧张时第一批被杀)。这套分级的假设是：服务的资源需求相对稳定，可以被一个固定档位描述。Agent 负载把三档同时击穿：需求在 185 MB 和 4 GB 之间跳，任何固定档位不是浪费就是危险；而且它是「杀不起」的有状态负载，BestEffort 直接出局。什么是 OOM 与 OOM killer？当系统内存耗尽，Linux 内核会挑一个进程杀掉以释放内存，这个行刑者叫

OOM killer。它不问青红皂白，按启发式评分挑「最该死」的进程——对 Agent 容器，往往就是那个积累了十分钟上下文的主进程。什么是「回收」（reclaim）与 GC 压力？memory.high 触发时内核会逼 cgroup 里的进程释放内存页；对 Node.js 这类运行时，释放内存意味着更频繁地跑垃圾回收，CPU 被白白消耗，Agent 响应变慢——软限流的代价悄悄转嫁到了运行时健康上。「子容器资源域」则是论文给出的方向性答案：在容器内部、以工具调用为边界建立更细的 cgroup 层级，让粒度匹配需求的自然边界。

Insight#1 memory.max 的两难（浪费 90%+ 或 OOM）本质是一个信息论问题：容器级接口一次只能表达一个数字，而需求是一个跨三个数量级的分布——表达能力的欠缺，不是调参能补的。

Insight#2 「多开容器细化粒度」被镜像大小否决，说明粒度问题有上下两个边界：下界是工具调用的需求波动，上界是容器创建的成本——可行解只能存在于容器内部，这个夹逼论证非常干净。

Insight#3 Kubernetes QoS 把 CPU 与内存捆绑分档，恰好踩中 Agent 负载 CPU-内存弱相关（均值 -0.39）的痛点：云原生资源模型的「资源包」抽象在这类负载上从根基处错配。

9.3 错位二：响应速度——秒级突发 vs 毫秒到分钟级的回路

Agent 的内存突发只持续 1-2 秒、变化率高达数 GB/s，且发生时刻不可预测（§ 3.3-3.4）。要在这种负载上有效干预，控制器必须「实时、内核速度」反应。现有方案的反应速度分布在什么量级？我们逐一过堂。

第一类是 PSI 驱动的用户态方案。PSI（Pressure Stall Information，压力停滞信息）^[35]是 Linux 内核 2018 年引入的资源压力度量接口（由 Meta 的 Johannes Weiner 提出，本文用它代表「内核信号 → 用户态守护进程」这一技术路线）。基于 PSI 的 oomd^[26]（Meta 开源的用户态 OOM killer，在内存压力超阈值时按策略杀 cgroup）和 systemd-oomd^[31]（systemd 内置的同类守护进程）会监控 cgroup 的压力指标，超阈值后执行杀进程或触发 swap。这套方案有两个隐含假设：(a) 压力信号出现后有足够的时间窗口做决策；(b) 杀掉或迁移是可接受的降级路径。两条对 Agent 全部失效：从 PSI 信号产生，到用户态守护进程收到、决策、再写 cgroup 控制文件，整个回路要数十毫秒——对于 1-2 秒的突发看似够用，但突发以 3 GB/s 的速度攀升时，几十毫秒内内存已上涨数百 MB，往往「响应到达时突发已越过不可逆点或已自行结束」。且 PSI 是容器级聚合指标，无法把压力归因到具体某次工具调用——知道「着火了」，不知道「哪个房间」。

第二类是编排层的垂直扩缩容。Kubernetes VPA（Vertical Pod Autoscaler，垂直 Pod 自动扩缩容）^[20]根据历史使用自动调整 Pod 的资源请求与上限，但生效需要重启 Pod；in-place resize（原地调整）^[18]是 K8s 社区 2024 年推进的增强提案（KEP-1287），允许不重启调整，但目前仍是 alpha 特性且生效粒度在分钟级。Pod 重启级到分钟级，对 1-2 秒的突发慢了 2-4 个数量级，执行期内的调整无从谈起。雪上加霜的是：反应慢一旦酿成 OOM，Agent 容器的高重启成本（§ 4.1：初始化占 31-48%）会把每一次漏判放大成分钟级的损失。

【零基础知识补给站】

什么是控制回路（control loop）？一切自动控制的基本结构：传感器观测 → 决策器判断 → 执行器干预，循环往复。回路有两个关键指标：延迟（从异常出现到干预生效要多久）和粒度（观测与干预的精细程度）。PSI + oomd 这条回路是：内核统计压力 → 写入 /proc/pressure → 用户态守

护进程轮询读取 → 策略判断 → 写 cgroup 文件执行。每一环都要跨越用户态/内核态边界，累计延迟数十毫秒，且观测粒度是整个 cgroup。什么是 TMO（Transparent Memory Offloading，透明内存卸载）^[36]？它是 Meta 在 ASPLOS 2022 发表的系统，用 PSI 信号把「冷」内存页异步换到 cheaper 存储（如 NVMe 或压缩内存），为数据中心省下 20–32% 的内存——本文把它列为反应式控制的一员。但 TMO 的服务对象是「缓慢变化的内存压力」，它的回收动作是分钟级预热、秒级执行，同样追不上 3 GB/s 的突发分配。VPA 与 HPA 的区别也顺带澄清：HPA（水平扩缩容）靠「加实例」应对流量，适合无状态服务；VPA 调整单个 Pod 的规格。Agent 是有状态单实例负载，HPA 用不上，VPA 又太慢——云原生的两大自动扩缩容工具在 Agent 场景双双缺席。把这三种方案的反应时间排一排：oomd 数十毫秒（但假设可杀）、TMO 秒到分钟、VPA 重启级到分钟——而需求是亚秒级。错位不在系数，在数量级。

- Insight#1** 响应速度错位的根源是「控制回路的架构位置」：用户态守护进程方案在原理上就存在跨态延迟下限，面对 GB/s 级突发，这个下限就是天花板——把决策移进内核不是优化，是唯一解。
- Insight#2** PSI 的「无归因」缺陷常被忽视：即便反应够快，容器级聚合信号也不支持「只惩罚那次 pytest」的精细操作——速度与粒度两个错位在 PSI 上叠加出现。
- Insight#3** 「突发可能已自行结束」揭示了慢响应的另一种失败模式：不仅救不了急，还可能误伤——干预落在突发回落之后，等于对已经恢复的负载施加无谓惩罚（笔者推断）。

9.4 错位三：适应性——历史预测 vs 不可预测的有状态执行

第三种错位最深。传统云负载大体确定，因此「以史为鉴」成为主流：Google Borg^[32]用历史利用率指导分配，Autopilot^[29]用历史 P95 自动生成资源推荐。Agent 负载沿三条轴线同时违反这个假设：其一，跨任务需求差一个数量级（197 MB–4 GB），VPA 给出的推荐值注定不是太高就是太低；其二，同一任务重复运行产生不同需求（ $1.8\times$ 时长方差），输出 token 数与峰值内存几乎零相关（ $r = -0.14$ ），历史百分位在统计上失去意义；其三，单次执行内部，重试循环造成渐进式内存爬升——需求在一次运行之内都在漂移，遑论跨运行。

预测失效后，传统方案的最后兜底是「kill-and-restart」（杀掉重启）。论文指出这对 Agent 是**三重惩罚**：第一，恢复慢——多 GB 容器的冷启动占任务总时长的 31–48%，比 serverless 冷启动高几个数量级；第二，状态丢——OOM 杀死进程即销毁数分钟积累的进程内 LLM 上下文，与微服务不同，这种上下文无法检查点（checkpoint）、无法迁移；第三，重跑不收敛——同一任务重跑会走完全不同的解法路径（§ 3.4），重启不保证回到原来的解决方案，甚至可能更差。任何以「终止」为兜底手段的策略都要同时吃下这三记闷棍。

由此论文导出设计原则的转向：**有效的资源管理必须用「优雅降级」（graceful degradation）——限流（throttling）、冻结（freezing）——代替终止，并利用 Agent 能根据资源反馈调整行为的独特能力。**我们把论文 Table 2 完整重建如下，作为三大错位的总结。

	静态限额	反应式控制	预测式扩缩容
代表工具	memory.max/high、cpu.max [14]；K8s QoS [19]	PSI [35]；oomd [26, 31]；TMO [36]	VPA [20]；Autopilot [29]

	静态限额	反应式控制	预测式扩缩容
隐含假设	峰值已知；需求稳定	压力渐进；杀进程可接受	负载可重复；历史有效
Agent 现实	峰均比 15.4×；需求随工具语义剧变	1-2 秒突发；时机不可预测	同任务 1.8× 方差；杀 = 丢上下文
错位类型	粒度（Granularity）	响应速度（Responsiveness）	适应性（Adaptability）

表 2（重建自论文 Table 2）：现有资源管理工具与 Agent 负载的错位对照。

读这张表的方式是按列看「假设 vs 现实」的落差：每一列是一种方案家族，第二行是它赖以成立的假设，第三行是 Agent 负载用 § 3 的数据给出的反证，最后一行是错位的名字。三个家族、三条假设、三记反证——没有哪一列有修复的希望，因为它们失败的原因都在各自架构的公理层，而非实现层。

【零基础知识补给站】

什么是「优雅降级」？系统过载时不直接崩溃或杀死任务，而是降低服务质量继续运行：视频网站从 4K 降到 720P，交易系统从实时撮合降到排队处理。对应到 Agent 内存场景，降级手段有两档：限流（throttling）——让分配内存的进程变慢，争取回收时间；冻结（freezing）——cgroup v2 提供 cgroup.freeze^[14]，可以整棵子树暂停，等内存宽裕再解冻。两者都保住进程活着、上下文完整。什么是「检查点」（checkpoint）？把进程的完整内存状态存到磁盘，之后可以从断点恢复。批处理作业常用它容错。但 Agent 的上下文不仅在本地进程里，还在云端 API 的会话里（对 Haiku 而言），本地 checkpoint 无法重建云端状态——这是「进程内有状态 + 云端会话」混合架构带来的新困难，纯系统技术无解。回顾一下三个错位的逻辑关系：粒度错位是「空间」问题（控制域太大），响应速度错位是「时间」问题（回路太慢），适应性错位是「知识」问题（未来不可知）。三者正交，任何一个的解决方案都无法顺带解决另外两个——这就是为什么第 10 章的 AgentCgroup 需要三个独立机制分别对应，而不是一个银弹。最后记住这张 Table 2 的用法：它是评估任何「Agent 资源管理新方案」的试金石——先问它假设什么，再问假设在 15.4×、1-2 秒、1.8× 面前站不站得住。

Insight#1 三大错位的共同根源其实是「无归因 + 无协商」：现有系统既不能知道「这次压力是谁造成的」，也不能告诉负载「请收敛一点」——两个通信通道的缺失比任何单点技术差距都根本（笔者推断）。

Insight#2 三重惩罚中「重跑不收敛」最容易被低估：它意味着 Agent 的容错不仅是工程问题，还涉及任务成功率——资源事故会转化为质量事故，SLA 设计必须把两者联合考虑（笔者推断）。

Insight#3 「优雅降级代替终止」这个原则对 Agent 有独特的可行性：Agent 能读懂「你被限流了」的反馈并调整策略——传统负载听不懂，只能被杀。这是第 10 章双向协议的思想种子，也是全论文最有原创性的观察。

Insight#4 Table 2 隐含一个分级判断：粒度与响应速度错位理论上可用更硬核的系统技术（内核态、细粒度）解决，适应性错位则无法纯靠系统技术解决——必须引入负载侧的智能，这把「系统设计题」升级成了「系统×AI 协同设计题」。

第 10 章 设计总览：意图驱动的资源控制

第 9 章用「排除法」清空了牌桌：静态限额、反应式控制、预测式扩缩容三大家族全军覆没。本章对应论文 § 5 的开头部分与设计哲学，看作者如何在废墟上重建。AgentCgroup 的设计可以浓缩成一句话：**用三个机制——对症三个错位，并用「意图驱动」取代「历史预测」作为资源协调的认识论基础**。本章先讲机制对应关系，再讲为什么其他路都走不通（排除法论证），最后讲全文最亮的那个洞察：Agent 是第一种「能听懂资源反馈」的负载。

10.1 三大机制——对应三大错位

论文 § 5 开篇就给出了工整的对仗结构，三个机制分别回应 § 4 的三个错位：

- **细粒度资源域 (fine-grained resource domains) → 粒度错位**。用 cgroup v2 的层级结构，把每个 Agent 任务映射为一个 cgroup 节点，每次工具调用是它的子节点，从而在保持任务总预算的前提下实现「按工具调用」的资源约束。落地方式是一个透明的 bash wrapper：拦截每次 bash -c 调用，为这条命令创建临时子 cgroup (`tool_<pid>_<ts>/`)，把进程挪进去、收集每次调用的资源指标 (memory.peak、时长)、命令结束后回收 cgroup。全程不改 Agent 框架一行代码。
- **内核态 eBPF 执行 (in-kernel eBPF enforcement) → 响应速度错位**。控制逻辑直接挂在内核 cgroup 的执行点上，微秒级反应，没有用户态-内核态往返。CPU 侧用 `sched_ext`^[15] (Linux 内核的可扩展调度器类，允许用 BPF 程序定义调度策略，出错时自动回退到默认行为)；内存侧用 `memcg_bpf_ops` 的钩子 (如 `get_high_delay_ms`) 在工具调用 cgroup 越过硬/软限时施加自定义限流延迟。内存压力升高时按优先级分级执行：先 `memory.high` 限流，再 `cgroup.freeze` 冻结——不杀进程，保住 Agent 状态。
- **意图驱动的资源协调 (intent-driven coordination) → 适应性错位**。既然预测不了，就建立双向协议让供需双方直接对话。这个机制是本章 10.3 的主角，此处先记住它的位置：它替换掉的不是某个技术组件，而是整个「以历史为据」的控制范式。



示意图 D5：一次工具调用的资源控制生命周期——wrapper 拦截、子 cgroup 创建、eBPF 内核态执行、意图反馈闭环。

怎么读这张图：它把三个机制串进一次工具调用的完整生命周期。从左到右：LLM 发出 bash -c 请求；wrapper 透明拦截（机制一）并创建临时子 cgroup；命令在子 cgroup 内执行，期间 CPU 由 sched_ext 调度、内存由 memcg_bpf_ops 看守（机制二）；若发生超限，限流/冻结而非杀死；命令结束后指标被收集，必要时向 Agent 注入反馈（机制三）；子 cgroup 销毁，一切不留痕迹。看懂这张图，就看懂了 AgentCgroup 的骨架。

【零基础知识补给站】

什么是 eBPF？它是 Linux 内核的「可编程沙箱」^[23]：允许把一段经过内核验证器（verifier）安全检查的小程序动态加载进内核，挂到各种事件点（系统调用、网络包、cgroup 操作）上执行。它的革命性在于：以前要改内核行为得改内核源码、重新编译、重启机器；现在像装插件一样即插即用，且验证器保证程序不会搞崩内核。eBPF 因此成为近十年云原生基础设施（网络、可观测、安全）的底层引擎。什么是 sched_ext？它是 Linux 6.12 起引入的调度器扩展框架^[15]：CPU 调度策略这个内核最核心的组件，也可以用 BPF 程序定制了，而且出错时自动回退到内核默认调度器（fail-safe），不会因为策略写错把机器卡死。什么是 memcg_bpf_ops？它是一组尚在评审中的内核补丁^[41]（Hui Zhu 2026 年提交到 bpf-next 的 RFC 系列，本文用它实现内存侧的内核态执行），给内存控制器加上 BPF 钩子——比如当 cgroup 内存超过 memory.high 时，由 BPF 程序决定给这个进程「罚睡」多少毫秒。为什么这些技术恰好解 Agent 的题？第 9 章说响应速度错位的根源是「用户态回路太慢」，而 eBPF 把决策逻辑放进了内核——事件发生处就地判决，微秒级，正是 1-2 秒突发所需要的速度。

Insight#1 三机制与三错位的——对仗并非修辞巧合，而是一种论证策略：它把「系统设计是否合理」转化为可逐条验证的命题——每个机制都能指认它消灭的错位，没有多余零件。

Insight#2 bash wrapper 看似是最「土」的组件（一个 shell 脚本），却是整个系统的支点：它把「工具调用」这个语义事件翻译成「cgroup 操作」这个系统事件——语义与机制的接口全部集中在这一点上。

Insight#3 「限流→冻结→不杀」的降级阶梯把第 9 章的三重惩罚直接归零：进程不死，上下文就在；不重启，冷启动惩罚和重跑不收敛就都不发生。降级路径的设计价值不低于控制逻辑本身。

10.2 设计哲学：排除法——为什么不预测、为什么不杀、为什么不改框架

好的系统设计，一半在选择做什么，另一半在说清不做什么。AgentCgroup 的三条「不做」，每一条都对应前文已经付过学费的教训。

不预测——因为 § 3.4 已经证伪

最直接的路线是「用历史数据训练一个预测器，提前给任务配资源」。第 8 章的三组数据已经把它枪毙：同任务 $1.8 \times$ 方差、跨任务 $20 \times$ ($CV = 147\%$)、token 与峰值内存 $r = -0.14$ 、轮数与峰值内存 $r < 0.11$ 。预测器的输入（任务 ID、历史百分位、推理侧指标）与输出（本次运行的资源需求）之间不存在可学习的稳定映射。AgentCgroup 因此彻底放弃预测，改为「声明 + 纠正」：Agent 声明意图，声明错了由实时执行与反馈回路兜底（10.3）。

不杀——因为 § 4.3 算过三笔账

第二条歧路是沿用传统兜底：超了就杀，杀了重启。§ 4.3 的三重惩罚——冷启动占 31–48%、LLM 上下文不可检查点、重跑不收敛——表明这条路的代价是任务级失败，而非实例级抖动。所以 AgentCgroup 的降级阶梯止步于冻结，终止只在所有手段失效后作为「最后的优雅降级路径」，并且即便此时也要先把自然语言反馈注入 stderr，让 Agent 带着信息重试，而不是不明不白地死。

不改框架——因为部署面不可行

第三条歧路是「改 Claude Code 的源码，让它原生上报资源需求」。工程上这当然能做，但生态上不可持续：Agent 框架众多且快速演进，每个框架改一遍、每个版本跟一遍，维护成本无底洞。bash wrapper 方案的选择理由是**透明拦截、零侵入**：bash -c 是各框架执行工具命令的公共接口，在接口处设卡，对所有框架一视同仁、即插即用。仓库分析报告也证实了这一取舍：wrapper 以脚本形式替换容器内的 /usr/bin/bash，单次调用额外开销低于 5 ms，相对 4–6 秒的 Bash 命令均值可忽略。

【零基础知识补给站】

什么是「排除法论证」？在系统论文里，一个新设计要证明自己「必要」，光说「我能行」不够，还要证明「别的都不行」。AgentCgroup 的论证链是：预测路线被 § 3.4 的测量证伪（数据层面），kill-and-restart 被 § 4.3 的成本分析证伪（经济层面），改框架被部署生态证伪（工程层面）——三条路都堵死后，剩下的「意图驱动 + 内核态执行 + 透明拦截」就不是偏好，而是唯一通路。这种论证方式的强度取决于排除是否穷尽。读者可以自行检验：还有没有第四条路？比如「给所有任务无脑配 4 GB」——那是静态限额，已被粒度错位否决；「在框架外加一个旁路监控」——那是

反应式控制，已被响应速度错位否决。「透明拦截」为什么值钱？在平台工程里，「要求应用配合」是最贵的前提：要改代码、要排期、要跨团队协作。透明方案把接入成本降到「替换一个文件」，这意味着它可以在不通知任何 Agent 开发者的前提下全网部署——历史上 eBPF 本身、iptables、透明代理都靠这一特质赢得生态（笔者类比）。一句话总结设计哲学：**把智能留在 Agent 里，把执行放在内核里，把侵入性降到零。**

Insight#1 三条「不做」分别由数据、成本、生态三种不同性质的论据支撑——这种多层次的否定比单层次否定更难被反驳，是值得学习的论文论证结构。

Insight#2 「不改框架」的深层含义是接口考古学：bash -c 能成为控制点，是因为它是 Unix 五十年沉淀出的公共接口——好的系统研究常常是在老接口上发现新语义（笔者推断）。

Insight#3 放弃预测并不等于放弃智能：AgentCgroup 把「预测的负担」从系统侧转移给了负载侧（Agent 自己声明），这在 AI 时代是更聪明的分工——谁拥有信息，谁负责估计。

10.3 关键洞察：Agent 是第一种「能听懂资源反馈」的负载

全论文最有原创性的思想，藏在意图驱动协调的双向协议里。先看它的两个方向：

- **上行 (Agent → 系统)**：每次工具调用前，Agent 可以通过环境变量声明预期的资源需求，例如 `AGENT_RESOURCE_HINT="memory:high"`（要跑测试套件）或 `"memory:low"`（只是读个文件）；wrapper 把这些提示映射为该次调用的 `memory.high` 限额。声明是建议性的（advisory）——说不准也没关系，反馈回路会纠正。
- **下行 (系统 → Agent)**：当一次工具调用被 OOM 杀死或限流到无法恢复时，wrapper 把自然语言反馈注入 `stderr`——比如峰值内存用量和「建议缩小范围重试」——Agent 读到后可以用更省资源的方式重试。当内核态执行单独无法阻止违规时，这条意图通道是终止前的最后一级优雅降级。

这个协议为什么本质性地不同于传统 auto-scaling？传统自动扩缩容的控制回路是**系统单方面猜**：观测指标 → 拟合模型 → 调整配额，负载是被动的、无知的、甚至敌对的（你限流，它照样申请）。而 Agent 是 LLM 驱动的：它能理解自然语言反馈、能重新规划行动、能在下一轮主动声明需求。控制回路第一次从「猜测-强制执行」变成「**人机协作协商**」——系统说「内存紧张」，Agent 说「那我分批跑测试」。这种能力此前从未被任何资源管理系统利用过，因为此前的负载（函数、微服务、批作业）都「听不懂话」。本文作者团队在 AgentSight^[39]（同一团队 2025 年发表的 eBPF Agent 可观测系统，AgentCgroup 的原型正是扩展自它）中已探索过「系统层观测 Agent」；AgentCgroup 再进一步，从「观测」走向「对话」。

【零基础知识补给站】

什么是控制回路里的「声明式 vs 命令式」？命令式控制是系统直接下命令：「这个 cgroup 最多 512 MB」。声明式则是负载先说意图：「我接下来可能需要大内存」，系统再据此安排。Kubernetes 的请求值（request）就是一种粗糙的静态声明，AgentCgroup 把它细化到每次工具调用、且允许动态修正。为什么「听得懂反馈」是 Agent 独有的能力？想象对一个 nginx 进程说「内存紧张，请你省着点用」——它没有任何机制理解这句话。但 LLM Agent 的工作循环本身就是「读文本 → 推理 → 行动」：把资源反馈写成文本塞进它的输入流，它就会像对待测试报错一样认真对待，并在下一步行动中体现（拆小命令、换轻量方案、先清理再重试）。LLM 的「指令遵循」能

力意外地为操作系统打开了一条新的控制通道——这是 AI 能力外溢到系统领域的典型案例。「优雅降级」在这里有了新层次：传统优雅降级是系统内部的技术手段（限流、冻结），AgentCgroup 多了一层语义降级——告诉 Agent「为什么被限」，让任务在受限条件下仍有机会完成目标。降级的终点从「活下来」升级为「完成任务」。当然也要冷静：声明是 advisory 的，Agent 可能报错数；反馈的有效性依赖模型的推理质量。这个协议把「资源管理」部分地押注在「模型能力」上，这是它的新颖之处，也是它的风险之处。

Insight#1 双向协议的本质是把资源管理从「估计问题」变成「协商问题」：估计需要历史数据（§ 3.4 证明没有），协商只需要当下的通信——绕开了不可预测性的正面，而不是硬闯。

Insight#2 「能听懂反馈」让 Agent 负载在可控性上反而优于传统负载：一个可被说服的负载，比一个只能被镇压的负载有更大的调度空间——这为「软性、协作式」的多租户管理开辟了全新的设计维度（笔者推断）。

Insight#3 下行反馈用 stderr 注入是一个深思熟虑的载体选择：它是 Agent 框架已经会原样转发给 LLM 的通道——不需要教框架任何新协议，反馈就自动进入推理上下文，零侵入原则贯彻到了最后一公里。

Insight#4 声明 advisory + 反馈纠错的组合，实际上是一个「在线学习」结构：Agent 从一次次反馈中校准自己的声明精度。资源系统与 Agent 在共同进化，这比任何静态策略都更接近长期稳态（笔者推断）。

10.4 设计边界与诚实声明：这是一份 proof-of-concept

严谨的阅读必须包含对边界的阅读。论文与代码仓库都相当诚实地标注了原型的局限，我们逐条列出，作为后续章节评估的前提：

- **原型是 proof-of-concept**：用 C 语言基于 libbpf 与 BPF CO-RE 实现^[24]（libbpf 是 eBPF 程序的标准加载库，CO-RE 即 Compile Once – Run Everywhere，让 BPF 程序跨内核版本可移植），扩展自 AgentSight^[39]。评估仅限于「真实 trace 的 50 倍速回放」，论文 § 7 明确承认：尚未在生产规模的真实 Agent 负载上验证，表征研究本身也只覆盖一个框架（Claude Code）和一个基准（SWE-rebench）。
- **内核依赖是未合入主线的补丁**：memcg_bpf_ops 来自 Hui Zhu 2026 年 1 月提交到 bpf-next 的 RFC 补丁系列^[41]（LWN 有报道），目前仍在 upstream 评审中。评估用的是打过补丁的 Linux 6.19.0-rc5 内核。
- **覆盖面的留白**：原型只管 CPU 和内存；初始化开销、大镜像、重试累积这三座大山在论文结论中被明确列为开放问题，需要沙箱运行时与编排栈的协同优化。

【注意】

给读者的实用提醒：**memcg_bpf_ops 补丁尚未合入 Linux 主线**。这意味着 AgentCgroup 的内存侧能力今天无法在任何发行版内核上直接运行，需要自行编译打过补丁的内核；CPU 侧的 sched_ext 则要求内核 6.12+。如果你的目标是学习思想，这些门槛无碍；如果目标是生产部署，必须评估「维护定制内核」的长期成本，或关注补丁的 upstream 进展。另外，仓库文档（docs/REPRODUCING.md）中的部分路径与当前代码布局存在漂移，复现时以 agentcg/ 目录的实际结构为准。

【零基础知识补给站】

什么是 proof-of-concept (PoC, 概念验证)? 研究原型的常见形态: 用最短路径证明「这条路走得通」, 而非「这条路好走」。PoC 的可信边界在于: 它证明了机制的有效性 (比如 throttling 能把 OOM 生存率从 66% 拉到 100%, 见后续评估章节), 但不证明工程的可推广性 (千台机器、多个框架、对抗性负载下是否依然成立)。什么是「upstream 评审»? Linux 内核由全球开发者以邮件列表协作开发, 任何新功能要经过漫长的 RFC (Request for Comments, 征求意见) → 逐轮评审 → 合入子系统树 (如 bpf-next) → 最终进入主线 → 再等发行版打包, 周期以年计。memcg_bpf_ops 处在最早期 (RFC 阶段), 存在被大改甚至拒绝的可能。什么是 CO-RE? 早期 eBPF 程序依赖具体内核版本的内部结构布局, 换个内核就要重编译; CO-RE 通过 BTF (BPF Type Format) 元数据让程序在加载时自动适配当前内核, 「一次编译, 到处运行」——这让 eBPF 工具的分发真正可行。理解这些边界的意义: AgentCgroup 的贡献目前是「测量 + 设计 + 可行性验证」三件套, 而不是「开箱即用的产品」。它的价值在于指出了方向 (意图驱动、内核态、工具调用粒度), 把工程长征留给了后续工作——论文结论部分对此毫不讳言。

Insight#1 依赖未 upstream 的内核补丁是一把双刃剑: 它限制了今天的部署, 但也说明这项工作站在了内核演进的最前沿——如果 memcg_bpf_ops 合入主线, AgentCgroup 的路径将成为「标准能力」而非「私家改装」。

Insight#2 「trace 回放」评估的隐含缺口是反馈回路无法闭环: 回放的 Agent 不会真的根据 stderr 反馈改变行为——双向协议中最创新的那一半, 恰恰是目前评估最少的一半 (笔者推断)。

Insight#3 把初始化开销、大镜像、重试累积列为开放问题, 实际上勾勒出了一个完整的研究议程: AgentCgroup 解决了「执行中」的资源控制, 而「执行前」(镜像、启动) 与「执行策略」(重试) 仍是空地——读者若想跟进研究, 这三块是现成的选题。

第 11 章 机制一（粒度）：bash wrapper 与细粒度资源域

第 4 章讲过，AgentCgroup 要解的第一个错配是「粒度错配」（granularity mismatch）：agent 的资源需求以「工具调用」为单位剧烈波动——git status 平均只吃 13.5 MB，pytest 测试执行的 P95 内存尖峰却高达 518 MB——但所有现有控制手段（memory.max、K8s QoS^[19]）都只能在容器级设一个策略。本章拆解论文给出的第一个机制：用一个透明的 bash wrapper（包装脚本），在不动任何 agent 框架代码的前提下，把每一次工具调用装进一个独立的、用完即弃的 cgroup 资源域里。

11.1 思路：不动 Agent 框架，替换 /usr/bin/bash

要给「每次工具调用」建一个资源域，最直观的办法是改 agent 框架——在 Claude Code 或 OpenHands^[34] 的工具执行逻辑里插入 cgroup 操作。但这条路有三个致命伤：第一，每个框架都要单独改，生态里有几十个框架；第二，框架代码是闭源或快速迭代的，补丁难以维护；第三，框架开发者没有理由为资源控制买单。论文选择的切入点非常取巧：**不改框架，改框架必然经过的那扇门**。

这扇门就是 **bash -c**。论文 § 5 指出，bash -c 是 agent 框架执行工具的公共接口：当 LLM 决定调用 Bash 工具时，框架最终都会 fork 一个子进程执行 bash -c "某条命令"。论文的测量（§ 3.2）也支撑这个选择：Bash 工具占了工具执行时间的 80.6%（Haiku）到 98.8%（GLM），抓住了 bash -c 就抓住了绝大多数资源消耗的来源。

具体做法在仓库 agentcg/bash_wrapper.sh 的头部注释里写得很直白：把真正的 bash 改名为 /usr/bin/real-bash，然后把这个脚本安装为 /usr/bin/bash。此后系统里任何人调用 bash，先进 wrapper，wrapper 做完自己的事后再把控制权交给真 bash。文件第一行的 shebang 就是 #!/usr/bin/real-bash——连 wrapper 自己都是由真 bash 来解释执行的，形成一个小循环。

为什么说这种「透明拦截」优雅？因为它满足系统改造的三个理想性质：**零侵入**（agent 框架、LLM、工具链全都不知道发生了什么）；**零依赖**（不需要框架提供插件机制，只依赖一个几十年不变的 Unix 约定）；**可整体回滚**（把文件换回去就恢复原状，wrapper 还提供了 AGENTCG_DISABLE=1 的一键透传开关）。代价则是粒度被钉死在「bash -c 命令」上——框架内嵌的 Read/Edit 等轻量工具（平均不足 0.5 秒、几 MB 内存）不经过 bash，自然也不在控制范围内；好在论文测量显示它们本来就不是资源大头（笔者评注：这是一个合理的工程取舍，而非疏漏）。

【零基础知识补给站】

什么是 /usr/bin/bash？什么是 shebang？ 在 Linux 里，「运行一个脚本」其实分两步：内核读到文件开头形如 #!/usr/bin/bash 的「shebang」行，就用它指定的程序来解释这个文件。bash 本身也只是 /usr/bin 目录下的一个普通可执行文件，不是内核里焊死的零件。所以「替换 bash」在技术上就是「换掉 /usr/bin 下的一个文件」——这和换掉一个 App 没有本质区别。而 agent 框架执行工具命令时，通常调用类似 exec("bash", "-c", 命令) 的接口，操作系统按 PATH 找到 /usr/bin/bash——找到的就是 wrapper。只要框架不绕过 PATH 直接执行某个私藏的 bash 副本，拦截就 100% 生效。在容器里部署更简单：把 wrapper 用 bind-mount 盖在 /usr/bin/bash 上即可（仓库 README 就是这么做的）。

Insight#1 好的系统机制往往不是「加一个新组件」，而是「找到一个所有流量必经的窄口」。

bash -c 就是 agent 工作负载的窄口。

Insight#2 透明拦截的价值与被拦截接口的稳定性成正比：bash -c 的语义几十年不变，这比任何框架插件 API 都可靠。

Insight#3 粒度控制做不到全覆盖时，优先覆盖资源大头（Bash 占工具时间 80%–98.8%）是正确取舍。

11.2 wrapper 逐段拆解

agentcg/bash_wrapper.sh 全文只有 121 行，却完整实现了「建域、执行、记账、销域」四步闭环。下面分四段拆解。

第一段：放行与分流

```
# If disabled, pass through directly
if [ "${AGENTCG_DISABLE:-0}" = "1" ]; then
    exec "$REAL_BASH" "$@"
fi

# Non "-c" invocations pass through (interactive bash, source scripts, etc.)
if [ "$1" != "-c" ]; then
    exec "$REAL_BASH" "$@"
fi
```

这段做「哪些调用需要管」的分流。两个 exec 把控制权完整转交给真 bash（exec 是替换当前进程，不会多出一层进程开销）。第一个分支是人工急停开关；第二个分支把非 -c 的调用（比如交互式 shell、source 脚本）原样放行——因为这些不是「工具调用」，截住它们只会增加干扰。分流的依据只有第一个参数，判断成本几乎为零，这正是「wrapper 开销 <5ms」的原因之一。

第二段：创建临时子 cgroup 并迁入

```
TOOL_CG="${CGROUP_ROOT}/tool_$$$(date +%s%N 2>/dev/null || echo $*)"
IN_CG=0
...
if mkdir -p "$TOOL_CG" 2>/dev/null; then
    if [ "$MEM_HIGH" != "max" ] && [ -f "$TOOL_CG/memory.high" ]; then
        echo "$MEM_HIGH" > "$TOOL_CG/memory.high" 2>/dev/null
    fi
    if echo $$ > "$TOOL_CG/cgroup.procs" 2>/dev/null; then
        IN_CG=1
    fi
fi
```

这是整个机制的核心动作，对应论文 §5 的「ephemeral child cgroup (tool_<pid>_<ts>/)」。

目录名里的 \$\$ 是当前进程 PID，date +%s%N 是纳秒时间戳，两者拼起来保证全局唯一——这就是论文里 tool_<pid>_<ts>/ 的由来。cgroup v2 里「建组」就是 mkdir，「设限」就是往伪文件 memory.high 里写字节数，「把进程放进组」就是把 PID 写进 cgroup.procs。三步全是文件操作，没有系统调用之外的新机制。

请注意每一行末尾的 `2>/dev/null` 和 `IN_CG` 标志位：`mkdir` 可能失败（`cgroupfs` 没挂载、没权限），写 `memory.high` 可能失败，写 `cgroup.procs` 也可能失败。`wrapper` 对每一种失败都静默吞掉，只在全部成功时把 `IN_CG` 置 1；后面的记账和反馈全部以 `IN_CG=1` 为前提。换句话说，**cgroup 不可用时 wrapper 优雅退化为纯透传**，绝不会让工具调用因为控制层故障而跑不了。这与 `sched_ext` 的 fail-safe 设计哲学（§ 5，见第 12 章）一脉相承。

第三段：执行与记账

```
"$REAL_BASH" "$@"
EXIT_CODE=$?
...
PEAK_MEM=$(cat "$TOOL_CG/memory.peak" 2>/dev/null || echo "")
CURRENT_MEM=$(cat "$TOOL_CG/memory.current" 2>/dev/null || echo "")
...
printf '{"ts":%s,"pid":%d,"cgroup": "%s", "cmd": "%s", "exit": %d, "duration_ms": %d, "peak_mem": "%s",
"current_mem": "%s", "hint": "%s"}\n' \
... >> "$LOG_FILE" 2>/dev/null
```

命令本体由真 `bash` 执行，`wrapper` 在原地等待退出码。命令结束后，由于整个命令及其所有子孙进程都待在 `tool_xxx cgroup` 里，内核已经在 `memory.peak` 里替我们记好了「这次调用的内存峰值」、在 `memory.current` 里记好了当前占用——这正是论文 § 3 表征实验中 `per-tool-call` 峰值数据的采集来源。最后一行把命令预览（截断 200 字符并替换引号防 JSON 注入）、退出码、耗时毫秒数、峰值内存、资源意图标签打包成一行 JSONL 追加到日志文件（默认 `/tmp/agentcg_tools.jsonl`）。一句话总结：**wrapper 顺手把「资源观测」也做了**，表征研究与资源控制共用同一条数据通路。

第四段：销域

```
# --- Cleanup: move back to parent cgroup, remove child ---
if [ "$IN_CG" = "1" ]; then
    echo $$ > "$CGROUP_ROOT/cgroup.procs" 2>/dev/null
    rmdir "$TOOL_CG" 2>/dev/null
fi

exit $EXIT_CODE
```

收尾有两步且顺序不能反：先把自己迁回父 `cgroup`，再 `rmdir` 临时组——`cgroup v2` 规定组里还有进程时不能删除。`rmdir` 之后，内核自动释放该组的统计状态，下一次工具调用拿到全新的记账。最后 `exit $EXIT_CODE` 把工具的真实退出码原样传给框架，保证上层看到的语义和没有 `wrapper` 时完全一致。整个生命周期就是论文反复强调的「与工具调用边界对齐的 `cgroup` 域」：随调用而生，随调用而亡。

【零基础知识补给站】

/sys/fs/cgroup 是什么？为什么说「建组就是建目录」？ Linux 把很多内核对象伪装成文件系统暴露给用户，`/sys/fs/cgroup` 就是其中之一（`cgroupfs`，伪文件系统）。你在里面 `mkdir` 一个目录，内核就创建一个 `cgroup`；目录里自动出现 `memory.high`、`memory.max`、`cgroup.procs` 等「文件」，其实都不是磁盘上的真文件，而是内核参数的读写接口：读 `memory.peak` 等于问内核「这个组的历史峰值是多少」，写 `cgroup.procs` 等于告诉内核「把这个进程挪进来」。删除一个 `cgroup` 就是 `rmdir` 它的目录（前提是组内无进程、无子组）。理解「一切皆文件」这个 Unix 传统

，wrapper 的每一行就都顺理成章了。

Insight#1 细粒度资源域的成本低得惊人：建组、设限、迁入、销组全是文件操作，wrapper 实测每次调用开销不足 5ms。

Insight#2 优雅降级写进了每一行代码：任何一步失败都静默退化为透传，控制层的故障域与业务层完全隔离。

Insight#3 memory.peak 让「每次工具调用的峰值内存」成为免费可得观测数据——表征与控制共用一条数据通路，是本设计最漂亮的一笔。

11.3 AGENT_RESOURCE_HINT 的解析与映射

细粒度资源域只解决了「能分开管」，还需要回答「每个域管多严」。这就是上行意图通道的用武之地：agent 在发起工具调用前，可以通过环境变量 AGENT_RESOURCE_HINT 声明这次调用的预期资源需求（§ 5）。wrapper 中负责解析它的代码是一段朴素的 bash case 语句：

```
HINT="${AGENT_RESOURCE_HINT:-}"
MEM_HIGH="max"
case "$HINT" in
  memory:low)  MEM_HIGH=$((256 * 1024 * 1024)) ;;
  memory:medium) MEM_HIGH=$((1024 * 1024 * 1024)) ;;
  memory:high) MEM_HIGH="max" ;;
  memory:*[gG]) ... MEM_HIGH=$(( NUM * 1024 * 1024 * 1024 )) ... ;;
  memory:*[mM]) ... MEM_HIGH=$(( NUM * 1024 * 1024 )) ... ;;
  "") ;; # no hint, use default (max)
  *) ;; # unknown hint format, ignore
esac
```

映射表一共三档语义加两种显式写法：**memory:low** → **256 MB**（对应 git status、文件浏览这类轻量命令，论文实测 Git 操作平均只占 13.5 MB，256 MB 绰绰有余）；**memory:medium** → **1 GB**；**memory:high** → **max**（不设限，交给下层的内核机制兜底）；以及 **memory:Ng / memory:NM** 直接写字节数（如 memory:2g = 2 GB）。解析失败的兜底一律是 max——宁可不管，不可错管。声明最终被翻译成 memory.high 这一个内核接口：超限时进程被回收减速（节流），而不是被杀掉。

为什么是「建议性」（advisory）而非强制？论文 § 5 给的答案是：agent 是善意的协作者而非恶意租户，它声明意图是为了换取不被杀；声明不准时，由下行反馈通道来纠偏（见 13.2 节），而不是用硬墙惩罚。用 memory.high（软限，触发回收与节流）而非 memory.max（硬限，触发 OOM killer）正是这个哲学的技术体现：论文 § 2 强调 memory.high 是「不调用 OOM killer 的软节流点」，这与 § 4.3 「kill-and-restart 对 agent 是三重惩罚」的分析严丝合缝。

值得指出的是（笔者依据仓库核实）：当前 wrapper **只解析 memory 维度**，没有任何 CPU 维度的 hint；tool 级 cgroup 也只写 memory.high，不写 cpu.max。CPU 差异化完全靠 session 级权重（session_high:session_low = 150:50，见第 12 章）体现。这与论文的表征结论一致——内存而非 CPU 才是多租户并发密度的瓶颈（§ 3.3）——但读者应知道「意图协议」在代码里目前只有内存一条腿。

【零基础知识补给站】

环境变量为什么能当协议用？ 环境变量是每个进程自带的一组「键=值」字符串，子进程创建时默认继承父进程的全部环境变量。agent 框架执行 `AGENT_RESOURCE_HINT="memory:low" bash -c "git status"` 时，这个变量会顺着 fork 链条一路传到 bash（即 wrapper）手里，wrapper 用 `$AGENT_RESOURCE_HINT` 就能读到。它不需要任何网络协议、SDK 或注册中心，是 Unix 世界里最轻量的「调用方给执行方递纸条」的方式。**memory.high 与 memory.max 的区别：**high 是软限，超了会被内核回收内存、拖慢分配，但进程活着；max 是硬限，超了直接触发 OOM killer 杀进程。对「杀掉就丢失全部 LLM 上下文」的 agent 来说，这个区别就是生与死的区别。

Insight#1 意图协议的全部复杂度就是一个 case 语句——协议的价值不在机制复杂，而在「agent 愿意说、系统愿意信」这个新关系。

Insight#2 用 memory.high（软限）而非 memory.max（硬限）落地意图，是「优雅降级优于终止」这一全文主线在最底层代码里的体现。

Insight#3 当前实现只有内存维度的意图解析；CPU 意图是论文设计与仓库现实之间的一处明确差距。

11.4 cgroup v2 层级继承的妙用

细粒度域方案有一个隐藏的扩展性问题：如果每次工具调用都新建一个 cgroup，那挂载在内核里的 eBPF 内存控制程序岂不是要感知、甚至重新挂接到每一个新建的组上？每秒可能新建好几个组，重新挂接既慢又有竞争风险。论文 §5 用一句话化解了这个问题：「memcg_bpf_ops attached to the parent cgroup automatically applies to all child cgroups via cgroup v2 hierarchy inheritance, requiring no BPF code changes」——挂在父 cgroup 上的 BPF 程序，经由 cgroup v2 的层级继承，自动覆盖所有子孙 cgroup，BPF 代码零修改。

这是 cgroup v2^[14] 层级模型的红利。cgroup v2（2015 年由 Tejun Heo 设计的统一层级控制组机制，Linux 内核官方文档^[14]）把资源组织成一棵树：策略沿树向下生效，会计沿树向上汇总。AgentCgroup 的树是这样的（笔者根据仓库代码整理）：`/sys/fs/cgroup/agentcg/` 是根；其下 `session_high` 与 `session_low` 是两个会话级节点（daemon 建层级时给它们分别写 `cpu.weight` 150 与 50）；wrapper 每次工具调用在 session 下建一个 `tool_<pid>_<ts>/` 叶子。挂在 session 级的 `memcg_bpf_ops` 回调（`below_low`、`get_high_delay_ms`，见第 12 章）在子孙组被回收或超限时同样被内核调用——临时组的生灭对控制面完全透明。

层级继承还带来一个会计上的便利：`session` 级的 `memory.current` 天然等于其下所有 `tool_` 子组与框架基线进程之和。也就是说，「整棵树的总量预算」与「每片叶子的细粒度限额」在同一套机制里并存，这正是论文 §5 所说的「per-tool-call 约束的同时保持整体工作负载预算」。daemon 在初始化时给 `session_high` 写 `cgroup.subtree_control "+memory +cpu"`，就是显式声明「允许该节点之下再开带内存与 CPU 控制器的子组」——没有这一行，wrapper 的 `mkdir` 建出来的组是不带内存控制能力的空壳。

【零基础知识补给站】

什么是「层级继承」？用一个比喻理解。 把 cgroup 树想象成公司的预算体系：CEO（根 cgroup）给事业部（`session_high`）批了年度预算和管理制度（挂上去的 BPF 策略），事业部内部每立一个

项目（tool_子组），不需要再向董事会申请制度——制度自动覆盖新项目，项目的开销也自动汇总进事业部的总账。cgroup v2 的 subtree_control 则像「授权书」：父节点必须在 cgroup.subtree_control 里写明 +memory +cpu，子节点才有资格使用对应的控制器，防止任意一层随意开无限额的口子。cgroup v2 相对 v1 的最大改进就是把原来多套各自为政的层级（CPU 一棵树、内存一棵树）统一成一棵树，继承语义因此清晰统一^[14]。

Insight#1 临时 cgroup 生灭的高频性与 BPF 挂接的低频性之间的矛盾，被 cgroup v2 的层级继承天然消解——这是「选对抽象」的胜利。

Insight#2 层级树同时给出两种视角：叶子是细粒度限额单位，父节点是总量预算与汇总单位，一套结构两种用途。

Insight#3 subtree_control 是层级继承的安全阀：控制能力的下放是显式授权，而非默认可得。

第 12 章 机制二（响应性）：内核内 eBPF 执行

第 11 章解决了「管多细」，本章解决「管多快」。论文 § 4.2 的响应性错配说得很清楚：agent 的内存突发只持续 1–2 秒、变化率高达数 GB/s，而用户态控制回路（PSI 信号 → daemon 决策 → 写 cgroup 文件）需要数十毫秒，等它反应过来，突发已经过去或者 OOM 已经发生了。

AgentCgroup 的答案是论文的第二个机制：**把控制逻辑用 eBPF 直接放到内核的执行点上**，反应时间从毫秒级降到微秒级。CPU 侧用 `sched_ext`^[15]，内存侧用 `memcg_bpf_ops`^[41]。

12.1 CPU 侧：sched_ext 与 scx_flatcg

`sched_ext`^[15]（Tejun Heo 等人设计的可扩展调度器框架，Linux 内核官方文档^[15]）允许用 BPF 程序定义 CPU 调度策略，内核在选进程、入队、出队等关键点位回调 BPF 代码；一旦 BPF 程序出错或超时，内核自动回退到默认调度器（fail-safe reversion）。AgentCgroup 的 CPU 侧实现是 `agentcg/scheduler/scx_flatcg.bpf.c`（954 行）。

先说一个容易被论文措辞掩盖的事实（笔者依据仓库核实）：**scx_flatcg 不是为 agent 新写的调度器**，它是内核 `sched-ext` 官方示例调度器 `flatcg` 的移植，文件头注释完整保留了原版的设计说明。它的核心算法叫「层级权重折叠」（flattening the cgroup hierarchy）：传统层级调度要一级一级地分配 CPU，`flatcg` 则把每个活跃 cgroup 沿层级复合出一个总权重，摊平成单层竞争。头注释里的例子很直观：B 在父级 A 内分得 1/2，A 在顶层分得 1/3，B 的最终份额就是 $1/2 \times 1/3 = 1/6$ ：

```
* For example, B is competing against C and in that competition its share is
* 100/(100+100) == 1/2. At its parent level, A is competing against D and A's
* share in that competition is 100/(200+100) == 1/3. B's eventual share in the
* system can be calculated by multiplying the two shares, 1/2 * 1/3 == 1/6.
...
* The scheduler first picks the cgroup to run and then schedule the tasks
* within by using nested weighted vtime scheduling by default.
```

调度循环是清晰的两级结构：**先选 cgroup，再选任务**。第一级，所有活跃 cgroup 按 `cvtime`（cgroup virtual time，虚拟时钟）排进一棵 BPF 红黑树（`cgv_tree`，带自旋锁），每次取 `cvtime` 最小的组服务；服务完一个时间片就按下式「预付」虚拟时间：

```
/* As the cgroup may have more tasks, add it back to the rbtree. Note
 * that here we charge the full slice upfront and then exact later
 * according to the actual consumption. This prevents lowpri thundering
 * herd from saturating the machine.
 */
bpf_spin_lock(&cgv_tree_lock);
cgv_node->cvtime += cgrp_slice_ns * FCG_HWEIGHT_ONE / (cgc->hweight ?: 1);
cgrp_cap_budget(cgv_node, cgc);
bpf_rbtrees_add(&cgv_tree, &cgv_node->rb_node, cgv_node_less);
bpf_spin_unlock(&cgv_tree_lock);
```

这行 `cvtime += slice × 基准权重 / hweight` 是权重公平的精髓：权重 `hweight` 越大，每服务一个时间片虚拟时钟涨得越慢，下次就越快再被选中——红黑树最左节点永远属于「最吃亏」的组。`hweight` 由父级权重与孩子权重之和复合而来（代码第 236–239 行）。第二级，组内任务再按 `dsq_vtime` 做加权虚拟时间调度（-f 选项可切 FIFO）。所有状态存在 BPF maps 里：每 cgroup

的 weight/hweight/活跃数放在 BPF_MAP_TYPE_CGRP_STORAGE（一种生命周期跟随 cgroup 的存储），红黑树节点用 16384 项的 HASH map 暂存。

那么「agent 优先级」体现在哪？答案是：**调度器本身完全不认识 agent**。优先级是 daemon 启动时用两行 cgroup 文件写入体现的（agentcgroupd.py）：

```
# Set CPU weights (higher = more CPU time)
cgroup_write(high, "cpu.weight", "150")
cgroup_write(low, "cpu.weight", "50")
```

150:50 即 3:1 的 CPU 时间份额比——论文宣称的「优先调度延迟敏感的工具调用」，在代码里就是标准的 cgroup v2 权重公平调度加上这一对权重值。而 fail-safe 由 sched_ext 框架免费赠送：BPF 调度器崩溃或 verifier 拒绝时，内核自动退回默认调度器，机器不会失去调度能力^[15]。

【零基础知识补给站】

什么是 struct_ops？什么是 cpu.weight？什么是红黑树 vtime 调度？ struct_ops 是 eBPF 的一种挂接方式：内核某个子系统（这里是调度器、内存控制器）定义一组「回调函数签名」，BPF 程序按签名实现其中几个函数，注册进去，内核在对应事件发生时调用——相当于给内核装插件。cpu.weight 是 cgroup v2 的 CPU 份额接口，默认 100；两个组权重 150:50，竞争时就按 3:1 分 CPU。「红黑树 vtime 调度」的直觉是：每个组有一把虚拟时钟，跑过就往前走，权重越大走得越慢；调度器永远选时钟最慢（最吃亏）的那个，长期下来各组得到的 CPU 就严格按权重比例。红黑树是一种能保持「找最小值」操作极快（ $O(\log n)$ ）的二叉搜索树，是调度器的经典数据结构。

Insight#1 AgentCgroup 的 CPU 侧创新不在调度算法，而在「把现成的 sched_ext 示例调度器接进了 agent 会话的权重体系」。

Insight#2 优先级 = cpu.weight 150:50 这一对数字；读懂这点，就把论文的「优先级调度」祛魅成了标准内核机制。

Insight#3 fail-safe 回退让「用 BPF 改调度器」从高危实验变成可上线特性——这是 sched_ext 相比内核模块的根本进步^[15]。

12.2 内存侧：memcg_bpf_ops

内存侧是论文真正的亮点，也是工程上最激进的部分。memcg_bpf_ops^[41]（Hui Zhu 提交至 Linux 内核邮件列表的 RFC 补丁系列，为内存控制器增加 BPF 钩子，LWN 报道^[41]）在内存回收路径上开了几个 struct_ops 回调口，其中两个被 AgentCgroup 用到：**below_low**（内核问：这个组是否应被视为低于 memory.low，从而免受回收？）和 **get_high_delay_ms**（内核问：这个组超过 memory.high 了，该罚它睡眠多少毫秒？）。实现位于 agentcg/memcg/memcg_priority.bpf.c（209 行）。

整个机制是一条三拍子的逻辑链：**探测 → 触发 → 双向动作**。第一拍「探测」：一个 tracepoint 处理函数挂在内核的 memcg/count_memcg_events 事件上，只统计 HIGH 优先级 cgroup 的缺页（page fault）次数，做 1 秒滑动窗口累加：

```
SEC("tp/memcg/count_memcg_events")
int handle_count_memcg_events(struct trace_event_raw_memcg_rstat_events *ctx)
{
    /* Only count page faults from HIGH priority cgroup */
```

```

/* PGFAULT = 23 in the kernel enum */
if (ctx->id != local_config.high_cgroup_id || ctx->item != 23)
    return 0;
...
if (current_ts - data->window_start_ts < ONE_SECOND_NS) {
    data->sum += ctx->val;
} else {
    data->window_start_ts = current_ts;
    data->sum = ctx->val;
}
if (data->sum > local_config.threshold) {
    bpf_map_update_elem(&trigger_ts_map, &key, &current_ts, BPF_ANY);

```

为什么是缺页？因为进程申请内存时，真正花时间的动作是「缺页 → 内核分配物理页」，缺页速率的激增是内存压力最早、最灵敏的先行指标。窗口内缺页数超过 threshold，就往 trigger_ts_map 写下当前时间戳——这等于宣布进入 1 秒「保护窗口」。

第二、三拍是保护窗口内的双向动作，各由一个 struct_ops 回调完成。对 HIGH 侧，below_low_impl 在窗口内返回 true，让内核把 HIGH 组当作「低于 memory.low」处理——低于 memory.low 的组在回收路径上享有豁免权：

```

SEC("struct_ops/below_low")
bool below_low_impl(struct mem_cgroup *memcg)
{
    if (!local_config.use_below_low)
        return false;
    if (need_protection()) {
        ...
        return true;
    }
    return false;
}

SEC("struct_ops/get_high_delay_ms")
unsigned int get_high_delay_ms_impl(struct mem_cgroup *memcg)
{
    if (local_config.over_high_ms && need_protection()) {
        ...
        return local_config.over_high_ms;
    }
    return 0;
}

```

对 LOW 侧，get_high_delay_ms_impl 在窗口内返回配置值 over_high_ms（实验用 50ms）：当 LOW 组超过自己的 memory.high 时，内核不再用默认的回收-重试节奏，而是让 LOW 组的分配进程睡指定的毫秒数——这就是论文里 239 次延迟触发（§ 6）的代码出处。两个 struct_ops 分别打包成 high_mcg_ops 和 low_mcg_ops，由 C 语言 loader（memcg_priority.c）挂到对应的 cgroup 上；配置（HIGH 组 ID、阈值、延迟毫秒）在挂接前由 loader 写进 BPF 程序的 .bss 段。

有一个隐式耦合值得点破（笔者依据仓库核实）：缺页计数是「HIGH 组活跃」的代理指标，而非全局内存压力；且 get_high_delay_ms 只在 LOW 组**已经超过**自身 memory.high 时才会被内

核调用。所以实验必须给 LOW 组设一个偏低的 memory.high (400 MB, 低于其 406 MB 峰值) 才能让节流被触发——机制的生效依赖一组精心配合的配置, 而非全自动。

【零基础知识补给站】

什么是缺页 (page fault) ? 为什么它能当先行指标? 程序申请内存时, 操作系统先「记账」不给真内存; 等程序第一次写这块内存时, CPU 发现没有对应物理页, 触发一次缺页异常, 内核这才分配物理页。所以缺页就像「工地真的开始动工」的信号——一个进程的缺页速率突然飙升, 说明它正在实打实地吞内存, 比「总占用已超阈值」要早半拍。**memory.low 又是什么?** 它是 cgroup v2 的保护线: 低于 low 的组在内存回收时免受影响。memcg_bpf_ops 的 below_low 回调相当于允许 BPF 程序临时「冒充」这条保护线——HIGH 组明明没有静态配置 memory.low, 却在压力窗口内被动态视为受保护。

Insight#1 探测-触发-双向动作的三拍子结构, 本质是把「优先级」从静态配置变成了随负载每秒刷新的动态状态。

Insight#2 缺页速率被选为压力先行指标, 是用最便宜的信号逼近「这个组正在抢内存」——信号便宜, 决策才能快。

Insight#3 机制有效但非自治: 缺页是代理指标、节流依赖 LOW 组预设偏低的 memory.high, 配置耦合是原型期的典型痕迹。

12.3 代价: 必须自编译内核

内存侧机制有一个无法回避的前提: memcg_bpf_ops **不在任何已发布的 Linux 内核里**。它目前是 Linux 内核邮件列表 (LKML) 上 review 中的 RFC 补丁^[41]。仓库以 agentcg/memcg/patches.mbox 的形式随源码分发了整套补丁——「[RFC PATCH bpf-next v6 00/12] mm: memcontrol: Add BPF hooks for memory controller」系列, 12 个补丁共 5571 行, 还包含 libbpf 侧配套的 bpf_map__attach_struct_ops_opts() 支持。要跑内存侧实验, 必须先给 bpf-next 内核树打补丁、自编译内核并重启进去 (论文 §6 的评估机跑的就是 Linux 6.19.0-rc5 + 这套 RFC 补丁)。

【注意】

这是 AgentCgroup 的部署门槛, 也是论文诚实标注的局限: README 写的前提「Linux kernel 6.12+ with sched_ext」只覆盖 CPU 侧 (sched_ext 已合入主线); 内存侧的 in-kernel 强制在可见的未来内都无法在生产内核上使用。仓库的对策是提供 cgroup v2 用户态回退后端 (见 13.3 节), 用标准 memory.low/memory.high 接口近似模拟 BPF 语义——功能可用, 但反应速度和精度都回退到用户态水平。评估一个基于未合入内核特性的系统论文时, 「上游 review 进度」应被当作与实验数据同等重要的风险指标。

从更长的视角看, 这未必是坏消息 (笔者评注): sched_ext 本身也经历了多年 RFC 才在 6.12 合入主线^[15]。AgentCgroup 选择在 RFC 阶段就基于 memcg_bpf_ops 构建原型, 实际上是在为上游 review 提供真实的使用场景与性能数据——论文中的 selftest 精度数据 (2000ms 配置延迟实测 $2.000 \pm 0.046s$, 相对误差 2.3%) 正是这类上游论证的典型素材。

【零基础知识补给站】

什么是「自编译内核」和 RFC 补丁? Linux 内核是开源的, 任何人都可以下载源码、打上别人的补

丁、编译出自己机器上运行的内核——但这通常需要几十分钟编译时间和一定的运维经验，而且新内核出问题要自己能兜底。「RFC」（Request For Comments，征求意见）是内核社区的惯例：作者把还没完全定稿的补丁发到邮件列表请大家评审，可能改很多版（这里已经是 v6）才被接受。在 RFC 阶段，接口随时可能变化，基于它写的代码也要跟着改。「patches.mbox」则是把一串补丁邮件打包成标准邮箱格式文件的分发方式，用 `git am` 命令就能按顺序应用。

Insight#1 AgentCgroup 内存侧的依赖链是：论文机制 → 未合入的 RFC 补丁 → 自编译内核——每一环都是部署成本。

Insight#2 提供用户态回退后端是原型系统面对现实约束的成熟做法：不回避门槛，而是把功能切成「完整版」与「降级版」。

Insight#3 上游 review 中的内核特性不是「不可用」，而是「收益与不确定性并存」；论文用原型数据反哺上游，是系统研究的正循环。

12.4 为什么不用用户态方案

一个自然的质疑是：内存压力检测和节流决策，用户态 daemon 也能做（读取 PSI、写 `memory.high`），为什么非要进内核？论文 § 4.2 给出的定量回答是：从 PSI 信号产生，到用户态 daemon 收到、做出决策、再写回 cgroup 控制文件，整个回路要数十毫秒；而 agent 的突发只有 1-2 秒，且内存变化率高达 3 GB/s——等用户态反应过来，「突发已经过去或已触发内核干预（OOM）」。对比之下，eBPF 钩子在内存回收路径上同步执行，决策延迟是微秒级（§ 5）。几十毫秒对微秒，是四个数量级的差距。

除了快，还有「准」。PSI^[35]（Pressure Stall Information，内核的压力停顿信息接口，Johannes Weiner 设计^[35]）是容器级的聚合指标，无法把压力归因到某一次工具调用；而 `memcg_bpf_ops` 的回调携带 `memcg` 参数，天然知道「是谁」在超限。对「tool 级粒度」这个论文核心诉求，用户态方案在原理上就拿不到足够细的信号。

那么改内核不行吗，为什么偏偏是 eBPF？（笔者综合论文 § 2 与背景知识推断）传统做法是写内核模块，但内核模块没有安全边界——一个空指针就能让整台机器崩溃，云厂商不敢为租户工作负载加载第三方模块。eBPF 通过 verifier^[23]（内核的静态检查器，Linux 内核文档^[23]）在加载前证明程序不会死循环、不会越界、不会泄露内核内存，是唯一同时满足「运行在内核里」「不会搞坏内核」「可动态加载卸载」三个条件的技术。论文 § 2 把这一点概括为「standardized interfaces 与 dynamic workloads 之间张力的解法」。

当然，「内核内执行」也不是免费的（笔者评注）：BPF 程序受 verifier 限制（无全局可变状态、循环必须有界），`scx_flatcg` 里红黑树操作要配自旋锁和 `stash map` 绕开限制；`memcg_priority` 里硬编码了「PGFAULT = 23」这样的内核枚举值，内核版本一变就要跟着改。这些工程摩擦在论文里看不到，但在仓库代码里俯拾皆是——这也是读源码对读论文的重要补充。

【零基础知识补给站】

用户态、内核态、以及「一次往返」的代价。 CPU 有权限级别：应用程序跑在用户态，内核跑在特权更高的内核态。用户态程序想请内核做事（读文件、分配内存），要触发一次「系统调用」切换特权级，来回有固定开销。一个用户态控制回路（如 `systemd-oomd`^[31] 或 `Meta oomd`^[26]，两者都是论文点名的用户态 OOM 管理器）做一次决策，至少要「内核发信号 → 用户态醒来 → 计算 →

系统调用写回」四次跨越，毫秒级延迟由此而来。eBPF 程序则被编译后加载进内核，在事件发生的同一现场同步执行，一次跨越都不需要。这就是「内核内执行」的全部秘密——不是更快地在路上跑，而是根本不上路。

Insight#1 响应性错配的本质是「控制回路的时间常数 vs 突发的时间尺度」：40ms 对 1-2s 看似够用，但对 3GB/s 的变化率已经太晚。

Insight#2 eBPF 的核心价值不是「快」，而是「快且安全」——verifier 让不可信方提供的内核逻辑第一次变得可接受^[23]。

Insight#3 内核内编程的摩擦成本（verifier 限制、枚举硬编码）真实存在；论文讲收益，仓库见成本，两者合读才是完整图景。

第 13 章 机制三（适应性）：意图驱动的双向协调

前两个机制解决了「管多细」和「管多快」，但都默认系统知道每次工具调用需要多少资源。论文 § 4.3 的适应性错配恰恰否定了这个前提：agent 的资源需求跨任务差 20 倍、同一任务跨运行差 1.8 倍、输出 token 数与峰值内存相关性接近零 ($r = -0.14$) ——历史统计在统计意义上失灵。AgentCgroup 的第三个机制因此不再预测，而是**协商**：建立一条 agent 与系统之间的双向意图通道。这是全文概念上最新颖的部分，也是仓库现实与论文宣称差距最大的部分——本章两者都讲。

13.1 上行通道：agent 声明资源意图

上行通道 (agent → system) 的协议简单到一句话：工具调用前，agent 在命令前面加一个环境变量——`AGENT_RESOURCE_HINT="memory:high" bash -c "pytest tests/"` 表示这次调用很重，`"memory:low"` 表示很轻 (§ 5)。wrapper 侧如何解析已在 11.3 节拆解过，这里要回答的是更根本的问题：**为什么是环境变量，而不是 API、配置文件或 IPC？**

（笔者综合论文与实现推断）第一个原因是**继承语义免费**：环境变量随 fork 自动传递给全部子孙进程，agent 框架不需要知道协议存在，变量自然穿透框架层到达 bash——协议对中间层透明。第二个原因是**零 API 负担**：任何框架、任何语言的 agent，只要能「执行一条 shell 命令」（这是所有 coding agent 的出厂能力），就已经实现了协议客户端；连 LLM 都只需在生成命令时多加一个前缀，不需要 function-calling 之类的新接口。第三个原因是**失败无害**：hint 缺失或格式错误时 wrapper 一律按 max 处理（见 11.2 节 case 语句的兜底分支），不声明的 agent 行为与未接入系统时完全一致——协议是纯粹的增量。

论文特别强调声明是建议性的 (advisory)：agent 可以低估，系统的反馈回路负责纠偏。这其实是把「准入控制」变成了「协商博弈」（笔者评注）：系统不因为 agent 报错数而惩罚它，而是让 agent 在「报错 → 被杀/被慢 → 收到反馈 → 下次报准一点」的循环中自己学会校准。这种设计之所以可行，前提正是论文反复强调的 agent 独有性质——它能理解自然语言反馈并重构执行策略，这是任何传统负载都不具备的能力。

【零基础知识补给站】

什么是环境变量继承？什么是 proof-of-concept？ 每个进程启动时都会从父进程那里拷贝一份环境变量表 (PATH、HOME 都是例子)，所以「在命令前写 KEY=VALUE」这种 shell 语法设置的变量会被该命令及其所有子进程看到——信息顺着进程树自动流动，不需要任何网络或文件。

proof-of-concept (PoC, 概念验证) 是系统研究的常用词：一个只为证明「这条路走得通」而做的原型，可以省略错误处理、兼容性和完整性，只要核心机制能演示、关键数字能测量。

AgentCgroup 论文明确自称 PoC (§ 5)，读它的代码时应用 PoC 的标尺，而不是产品的标尺。

Insight#1 环境变量协议的精髓是「利用 Unix 进程模型的默认行为做协议传输」——零成本、零依赖、失败无害。

Insight#2 advisory 而非 mandatory 的设计承认了一个现实：LLM 对资源需求的估计同样不准，所以系统必须自带纠偏回路。

Insight#3 判断一个协议好坏的标准之一：不参与的旧系统行为是否完全不受影响。
`AGENT_RESOURCE_HINT` 满足这条。

13.2 下行通道：把内核事件翻译成自然语言

下行通道（system → agent）做的事在系统研究里几乎没有先例：**把一次内核级资源事件，翻译成 LLM 能读懂、能据此调整策略的自然语言**。仓库中的实现位于 bash wrapper 的第 90–101 行：

```
if [ $EXIT_CODE -eq 137 ] && [ "$IN_CG" = "1" ]; then
    PEAK_MB=$((PEAK_MEM / 1024 / 1024))
    echo "[Resource] Command killed (OOM, exit 137). Peak memory: ${PEAK_MB:-unknown}MB." >&2
    echo "[Resource] The command exceeded the available memory budget." >&2
    echo "[Resource] Suggestions: run more targeted operations (e.g., specific test" >&2
    echo "  files instead of full test suite), reduce data size, or split into" >&2
    echo "  smaller steps. You can also request more memory by setting" >&2
    echo "  AGENT_RESOURCE_HINT=\"memory:<size>g\" before the command." >&2
fi
```

逐行看这段代码在做什么。触发条件是两个：退出码为 137，且本次调用确实在 cgroup 里运行。137 是 OOM killer 的签名：进程被 SIGKILL（信号 9）杀死时 shell 报告 128+9=137。满足条件后，wrapper 从刚读到的 memory.peak 算出峰值 MB 数，然后向 **stderr** 写入六行 [Resource] 前缀的文案：先报告事实（被杀、峰值多少），再给可执行的建议（缩小测试范围、减少数据量、拆成小步骤），最后教学式地告诉 agent 下次可以用 AGENT_RESOURCE_HINT 申请更多内存——建议里甚至包含了正确的命令语法。

为什么是 stderr？（笔者推断）因为 coding agent 框架的通用行为是把工具命令的 stdout/stderr 原样拼回 LLM 的上下文。写进 stderr，就等于写进 LLM 的「视野」；写进日志文件或返回值，LLM 看不见。这六行 echo 因此不是给人看的运维日志，而是**给 LLM 的 prompt 注入**——这是整条下行通道设计里最关键的一个字节：>&2。

闭环由此完成：agent 报告意图（上行）→ 系统按意图设限 → 意图报错导致 OOM → 系统用自然语言告知事实与替代方案（下行）→ LLM 在下一轮推理中选择更轻的策略或显式申请更多内存。论文 §5 称之为「final graceful degradation path before termination」——即使内核机制没能挡住违规，协商通道也能在终止之前给出最后一次体面退场的机会。

【零基础知识补给站】

exit code 137 是怎么来的？ Unix 进程退出时会给父进程留一个 0–255 的退出码。正常结束是自己 return 的数字；被信号杀死时，shell 约定报告 128 + 信号编号。SIGKILL（强制杀死，不可捕获不可忽略）的编号是 9，所以被 OOM killer 选中的进程留下 128+9=137。这就是为什么所有容器运维文档都教你看 137：它几乎等价于「这个进程因内存超限被杀了」。**stdout 与 stderr** 是每个进程的两条标准输出流：stdout 走正常结果，stderr 走错误与诊断。agent 框架通常两条都收集，所以 [Resource] 文案走 stderr 既能进入 LLM 上下文，又不污染命令的正常输出。

Insight#1 下行通道的创新不在技术（六行 echo），而在范式：内核事件第一次被翻译成了 LLM 可消费的语言。

Insight#2 stderr 是这个设计的命门字节——它利用了「框架会把工具输出带回给 LLM」这一普遍事实，零新机制完成触达。

Insight#3 反馈文案的质量决定闭环的效果：它必须既给事实（峰值数字）又给可执行的替代动作，否则 LLM 无从纠偏。

13.3 用户态 daemon：协调器与回退后端

前面三章的主角都在容器内（wrapper）或内核里（BPF），还有第三个组件坐在宿主机用户态：agentcg/agentcgroupd.py（351 行 Python）。论文 §5 对它的描述是「A lightweight user-space daemon manages cgroup lifecycle and policy configuration via shared BPF maps」。仓库里的现实更朴素：它是一个**协调器**，只做三件事——建 cgroup 层级、拉起三个 eBPF 子进程、把进程监控事件记日志。启动序列（start 方法）依次是：建 session_high/session_low 并写好 cpu.weight 150:50 与 subtree_control（见 12.1 节）；启动 scheduler/scx_flatcg -i 5；调用 create_memcg_controller() 自动选择内存后端；最后启动 process/process 进程监控器，把它的 JSON 事件流接到一个只记日志的循环上。代码注释说得很坦白：「With the bash wrapper active, per-tool-call cgroup assignment is handled by the wrapper itself. The daemon logs events for observability」——**daemon 不做任何动态策略决策**，所有策略参数都是硬编码的默认值。

比 daemon 本身更有意思的是它通过工厂函数选出的内存后端。agentcg/memcg_controller.py（399 行）定义了统一接口 MemcgController（attach/detach/poll/get_stats）和两个实现：BpfMemcgController 在有补丁内核时拉起 memcg_priority 二进制（完整版）；CgroupMemcgController 是**纯 cgroup v2 用户态回退后端**——它用三种标准内核信号检测压力（memory.events 里 high 计数增长、父 cgroup 的 PSI total 增长、父 cgroup 使用率 $\geq 85\%$ ），触发后向 HIGH 组写 memory.low = 总量 $\times 80\%$ 、向 LOW 组写 memory.high = 总量 $\times 50\%$ ，1 秒后恢复。

「回退后端」的意义值得展开（笔者评注）：它把 AgentCgroup 从「只有自编译内核才能跑的研究原型」变成了「任何 cgroup v2 系统都能用一部分的实用工具」。BpfMemcgController 与 CgroupMemcgController 之间是一张精心设计的语义对照表——below_low 回调 $\approx$ 临时写大 memory.low；get_high_delay_ms 回调 $\approx$ 临时压低 LOW 组 memory.high；缺页计数 $\approx$ PSI/事件计数三种代理信号。近似当然不完美：用户态反应慢、没有缺页级先行指标、节流粒度粗。仓库的实验记录也如实反映了差距：回退后端实测 HIGH 平均分配延迟 4.1ms 对 LOW 的 7.6ms，节流效果 1.85 倍——有效，但远不如内核内精确。这张「完整版 vs 降级版」的功能矩阵，是这个仓库工程成熟度的最好证明。

【零基础知识补给站】

daemon 是什么？工厂函数又是什么？ daemon（守护进程）是 Linux 里在后台长期运行、不与终端交互的进程，systemd-oomd^[31] 就是著名的内存管理 daemon。「工厂函数」create_memcg_controller() 是一种编程模式：调用者不直接 new 某个类，而是请工厂「探测环境、

帮我挑一个能用的」——这里它探测补丁内核与 BPF 二进制是否存在，存在就给 BPF 版，否则给回退版，上层代码完全不用改。**PSI** (Pressure Stall Information)^[35] 是内核导出的「进程因等资源而停滞了多久」的累计统计，用户态读它的增量就能知道「刚才内存压力有多大」，是 oom^[26] 等用户态方案的标准信号源。

Insight#1 把策略决策全部下放给 wrapper 与 BPF，daemon 退化为纯协调器——「简单即可靠」在这个架构里是刻意为之。

Insight#2 回退后端 + 工厂探测的模式，让论文机制在标准内核上保留了「可用子集」，是原型走向实用的关键一步。

Insight#3 daemon「只观测不决策」也提示读者：论文中的「runtime-adaptive policies」在当前代码里主要由静态配置承担。

13.4 论文承诺 vs 仓库现实的差距

本章开始时就预告过：意图驱动机制是「概念最新、差距最大」的部分。把论文 § 5 的句子逐条对到代码上（笔者依据仓库全文检索核实），有两处明确差距：

- **「被节流时也注入反馈」——未实现。**论文摘要与 § 5 都说，工具调用被 OOM 或 throttled（节流）时注入反馈。但 wrapper 中唯一的注入点是 exit 137 这一个分支；被 memory.high 节流（变慢但活着）的调用不会产生任何 stderr 反馈。也就是说，「反馈纠偏」闭环目前只在最极端的失败场景下闭合。
- **「cgroup.freeze 优雅降级」——未实现。**论文 § 5 写明了分级降级路径：「throttling via memory.high delays, then freezing via cgroup.freeze, rather than termination」。但全仓库（wrapper、daemon、memcg_controller）没有任何对 cgroup.freeze 文件的写入逻辑。冻结这一级目前只存在于论文文本中。

【注意】

指出差距不是贬低这项工作，而是示范阅读原型论文的重要方法论：**区分「系统设计」「已实现」「已评估」三个层次**。AgentCgroup 的 cgroup.freeze 属于「设计了、未实现」；下行反馈属于「实现了 OOM 分支、未实现 throttled 分支」；而 § 6 的 29% P95 改进属于「已实现且已评估」。三层混读会高估系统；只读代码又会低估设计的启发价值。诚实的阅读是把每一句宣称归到正确的层。

还有一点介于差距与取舍之间：AGENT_RESOURCE_HINT 协议只覆盖内存维度（11.3 节已述），CPU 意图不存在；且 hint 由 agent「自愿」提供，当前仓库没有任何机制引导 LLM 学会何时该报什么——论文设想的「agent 在反馈中学会校准」尚未在任何实验中被直接验证（笔者评注：这是把「意图驱动」从口号变成科学结论前必须补上的一环，也是 future work 中最有价值的方向之一）。

【零基础知识补给站】

为什么论文与代码会有差距？这正常吗？非常正常。系统论文写作与原型开发是并行推进的：论文描述的是「目标系统的完整设计」，代码是「在截稿前跑起来的那部分」。会议/期刊文化允许这种差距，前提是论文明确自称 prototype 或 proof-of-concept——AgentCgroup 在 § 5 末尾正是这么做的。成熟的阅读姿势是：把论文当「设计文档 + 部分验证」，把仓库当「实现进度的真相」。两

者差距本身也是信息——它告诉你哪些想法容易落地（wrapper 一周就能写完），哪些想法卡在工程细节里（freeze 的时序与恢复比想象中复杂）。

Insight#1 意图机制的「双向」目前只闭环了一半：上行已实现，下行只覆盖 OOM；throttled 反馈与 freeze 降级仍是文本承诺。

Insight#2 给论文宣称分层（设计/实现/评估）是阅读任何系统论文的通用防护技能，原型论文尤其需要。

Insight#3 差距最大的地方往往也是后续工作空间最大的地方——「agent 如何学会报准意图」是一个尚未被验证的核心假设。

第 14 章 评估解读（§ 6）

论文 § 6「Preliminary Evaluation」只有不到一页，却承载了系统部分的全部实证重量。它的每个数字都来自真实 agent 内存轨迹的回放，实验材料、配置与结果在仓库 `agentcg/memcg/multi_tenant_test/` 与 `EXPERIMENT_REPORT.md` 中可以逐一对账。本章按「怎么测、测出什么、该怎么看」三步解读。

14.1 实验设置

评估平台是一台 Intel Core Ultra 7 258V（4 核、16 GB RAM），内核为打过补丁的 Linux 6.19.0-rc5（bpf-next 树 + memcg struct_ops RFC 补丁^[41]）。注意这台机器与 § 3 表征实验的平台（24 核 128 GB 台式机）不同——评估刻意选了小内存机器，让「内存紧张」的场景在小尺度上成立（笔者推断：4 核 16GB 也更接近真实的单容器部署密度）。

实验不跑真实 LLM，而是 **trace replay（轨迹回放）**：取 § 3 采集的真实 agent 内存轨迹，由回放器（`trace_replay.py`）按采样序列真实地分配/释放内存（`bytearray` 按页 touch），以 50× 加速压缩时间轴——原来约 100 秒的会话压缩到约 2 秒墙钟。加速的实现毫无玄学（笔者依据仓库核实）：Python 的 `time.sleep` 间隔直接除以 50。

多租户配置是三个并发 cgroup：**HIGH** 会话跑 `dask/dask#11628` 的轨迹（峰值 421 MB，`memory.high = max`，即不预设软限）；两个 **LOW** 会话各跑一份 `sigmavirus24/github3.py#673` 的轨迹（峰值各 406 MB，`memory.high = 400 MB`）。三条轨迹合计需求约 1233 MB。BPF 策略下，HIGH 组缺页速率超阈值即进入保护窗口：HIGH 侧 `below_low` 生效免受回收，LOW 侧超过 `memory.high` 时被施加自定义延迟（12.2 节的机制）。对照组是 `no-isolation` 基线：同样三条轨迹、同样内存总量，但不启用任何 BPF 强制。

【零基础知识补给站】

什么是 trace replay？为什么回放可以当评估？ 回放就是「按剧本重演」：先把真实运行时每秒的内存占用记下来，评估时让程序按同样的节奏真实地吃掉/释放同样多的内存。对「内存控制器表现如何」这个问题，被控对象（内核回收路径、BPF 钩子）面对的内存压力与真实场景几乎无法区分——它们看到的只是 cgroup 里的字节数变化。回放换来的是**可重复性**：基线与 BPF 跑的是分秒不差同一份压力，差异只能归因于机制本身；而真实 LLM 每次行为都不同（§ 3.4 实测 1.8× 方差），根本无法做对照。50× 加速则让一次实验从小时级缩到分钟级，可以多跑多轮取统计。

Insight#1 评估设计的核心是「固定一切可固定的，只放开机制这一个变量」——回放是达成这一点的方法论。

Insight#2 三条轨迹、两种优先级、合计 1233 MB 对 1100/1300 MB 两种总量：场景是精心设计的「压力刚好差一点」的临界态。

Insight#3 用小内存机器做评估不是妥协而是设计：16GB 让隔离机制的效果在可控尺度上可观测。

14.2 结果一：OOM 存活率 66% → 100%

第一个场景是「紧内存」：总量只给 1100 MB，而三进程合计需求约 1233 MB——故意制造 133 MB 的缺口，看谁死。结果对应论文图 8(a)（下图为论文原图）：

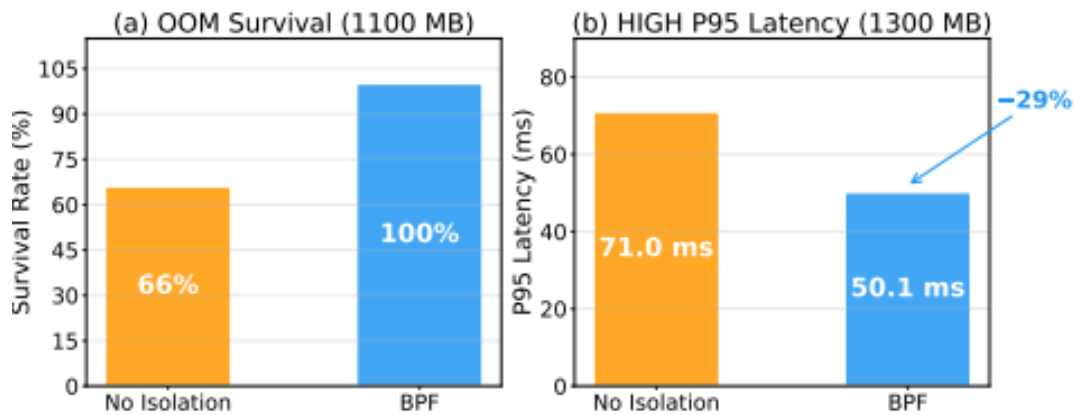


图 8（论文原图）：真实 agent 轨迹回放下的 BPF 强制评估。(a) 紧内存（1100 MB 总额 vs 约 1233 MB 需求）下的 OOM 存活率；(b) 中等内存（1300 MB）下 HIGH 优先级 P95 分配延迟。

图 8(a) 的读法：无隔离基线下，一个 LOW 进程被 OOM killer 杀死，三进程存活率 66%（3 取 2）；BPF 强制下，LOW 进程的内存分配被延迟压制了 **239 次**（论文 § 6 原文 239 delay triggers，即 12.2 节 get_high_delay_ms 的生效次数），挤出的内存让三个进程**全部**跑完，存活率 100%。被杀掉的那个 LOW 进程在真实场景里意味着什么？回到 § 4.3 的三重惩罚：容器冷启动占任务时间 31–48%、丢失全部已积累的 LLM 上下文、且重跑不保证收敛——一次 OOM 对 agent 不是重启，是任务失败。

代价也如实记录在仓库实验报告里（论文正文未展开，笔者补充）：LOW 进程的完成时间从 2.17s 延长到 4.40s（50× 压缩后的墙钟），约翻倍；而 HIGH 进程只慢了 +2.8%。这正是优先级机制想要的不对称：延迟被定向转移到低优先级租户身上，高优先级租户几乎无感。

【零基础知识补给站】

存活率 66% 是怎么算出来的？ 三个进程里死了一个，活下来 $2/3 \approx 66\%$ 。为什么是 LOW 死而不是 HIGH 或随机？Linux OOM killer 按 badness 评分挑受害者，大致是谁占内存多先杀谁；两个 LOW 进程轨迹峰值各 406 MB、且没有保护，自然首当其冲。「**239 次延迟触发**」意味着什么？回放全程约 2 秒墙钟（50× 压缩后），LOW 进程的分配动作在这 2 秒里被强制睡眠了 239 次——高频、小幅的「点刹」，而不是一次性的「急停」。这正是节流优于杀死的原因：进程始终在前进，只是变慢。

Insight#1 66% → 100% 的存活率差，本质是把「硬性的二选一牺牲」换成了「软性的延迟再分配」。

Insight#2 代价分布（LOW ×2 时长，HIGH +2.8%）证明优先级语义是真实的，而非两边一起变慢的平均主义。

Insight#3 239 次触发这个数字本身就是机制工作强度的直接证据——它证明收益来自高频微干预，而非一次性兜底。

14.3 结果二：P95 降 29%，开销可忽略

第二个场景放宽到「中等内存」（总量 1300 MB，不再有人死），改测**尾延迟**：图 8(b) 显示 HIGH 进程的内存分配 P95 延迟从 70.97ms 降到 50.14ms，**下降 29%**；P99 从 192ms 降到 184ms（仓库数据，论文未引用）。机理链条是：LOW 被节流 → 全系统内存竞争减弱 → HIGH 的分配路径上回收与缺页处理变快 → 尾部延迟下降。29% 这个数字也被写进了摘要，成为系统部分的招牌结果。

机制自身开销用第三个实验回答：内存充足（2000 MB）时跑 300 次 10MB 分配的微基准，BPF 钩子的介入让 P50 延迟从 0.667ms 变为 0.669ms，**仅 +0.3%**；三个进程的总完成时间甚至 **下降 1.1%**（在测量噪声范围内，应读作「无显著影响」）。第四个实验测精度：内核 selftest 配置 2000ms 延迟，实测 $2.000 \pm 0.046s$ ，相对误差 2.3%——证明 `get_high_delay_ms` 下发的毫秒数被内核忠实地执行，机制是「指哪打哪」的。

把四个数字放在一起看，论文的论证结构很标准：**有效性**（存活率 100%、P95 -29%）+ **代价**（P50 +0.3%、总时长 -1.1%）+ **精度**（2.3% 误差）。这是系统论文评估的经典三段式：证明有用、证明便宜、证明可控。在「初步评估」（preliminary）的自我定位下，这组数字是完整且自治的。

【零基础知识补给站】

P50、P95、P99 是什么？ 把一次实验里所有延迟样本从小到大排队，排在 50%、95%、99% 位置的值就是 P50（中位数）、P95、P99。P50 告诉你「典型情况」，P95/P99 告诉你「倒霉的时候有多惨」。在线服务和调度系统尤其看重尾延迟，因为用户耐心由最差的那批体验决定；对 agent 来说，工具调用的 P95 直接决定任务总时长的波动。「**相对误差 2.3%**」则是实测值与目标值的偏离比例： $|2.000 - 2.000 \times (1 \pm 0.023)|$ 的范围内， $\pm 0.046s$ 的抖动完全在误差带内——说明 BPF 下发的延迟指令被内核精确执行。

Insight#1 P95 -29% 的机理不是「BPF 让内存变快」，而是「把竞争从 HIGH 的分配路径上挪开」——收益来自排除干扰而非加速自身。

Insight#2 +0.3% P50 与 -1.1% 总时长说明钩子几乎零成本：机制的钱全花在「保护窗口」那 1 秒里，平时分文不取。

Insight#3 selftest 2.3% 误差是被低估的结果：它证明这套机制不是统计意义上的平均有效，而是逐次精确可控。

14.4 批判性评注

以下评注均为笔者观点，意在给这组漂亮数字标定正确的置信区间——它们扎实，但边界清晰。

- **只有 3 条 trace、1 种优先级拓扑。** 全部结论建立在 `dask#11628 + 2 × github3.py#673` 这一个组合上。三条轨迹的峰值（421/406/406 MB）与限额（1100/1300 MB）之间的距离显然是精心调过的：缺口 133 MB 恰好「基线必死一个、BPF 恰好能救」。这不构成数据操纵——临界场景本就是压力测试的合理选择——但 29% 与 66% → 100% 都应理解为「在精心构造的内存压力场景下取得」，不能直接外推到任意负载组合。

- **回放 ≠ 活的 agent。** 回放只复现内存曲线，不复现因果：真实场景里，LOW 被节流后 agent 会改变后续行为（比如重试、换策略），这些反馈效应全部缺席。§ 6 测的是「机制对

固定压力曲线的响应」，不是「机制对端到端任务成功率的影响」。论文把评估限定为 preliminary，正是对此的诚实标注。

- **缺少中间对照。**对照组只有 no-isolation 与完整 BPF 两档。而仓库自己实现的 CgroupMemcgController（用户态回退后端，13.3 节）天然构成第三档：「纯用户态 + 标准 cgroup v2」能达到 BPF 版的几成效果？仓库实验日志显示它能做出 $1.85\times$ 的节流差异，但论文未将其纳入对照。有这一档，「为什么必须进内核」这个论文的核心主张才能得到定量的支撑（笔者评注）。

综合来看（笔者总结）：§ 6 证明了机制在目标场景下「存在且有效」，这是初步评估的本分；它没有证明、也不声称证明「优于一切替代方案」或「对任意负载有效」。读者应把 29% 当成「机制上限的一次取样」，而不是「部署后的预期收益」。

【零基础知识补给站】

什么是「对照实验」？为什么中间档重要？ 对照实验要求：除被测变量外，其余条件完全一致。本文的对照（no-isolation vs BPF）回答了「BPF 机制有没有用」，但不能回答「BPF 是不是必须的」——因为也许一个只写标准 cgroup 文件的用户态脚本也能达到类似效果。要回答后者，需要加入「中间档」：功能逐档递减的对照组。中间档在系统评估里极其常见，例如「无优化 / 只开优化 A / 全开」的消融实验。判断一份评估的说服力，先数它的对照档位能不能支撑它想回答的那个「为什么」。

Insight#1 临界场景是压力测试的合法武器，但结论的适用范围以场景构成为边界——外推需谨慎。

Insight#2 回放证明了机制，却没有证明闭环；「活的 agent」评估是本文之后最重要的下一步。

Insight#3 缺少用户态中间档，使「必须进内核」目前主要靠 § 4.2 的定性论证支撑，而非 § 6 的定量证据。

第 15 章 相关工作与参考文献关联分析

论文只有 8 页正文，却列了 41 条参考文献。这 41 条不是装饰，而是一张精确的「论证地图」：每一条都在论文的推理链上占据一个位置。本章不按编号顺序，而按**论证功能**把它们分成五组，逐组讲清「这篇文献是什么、本文用它论证什么、引用逻辑如何咬合」。读完本章，你应能仅凭参考文献表就重建出这篇论文的学术坐标。

15.1 传统负载测量三剑客：方法论的直系血统

论文 § 3 开头有一句话值得划线：「Designing effective resource controls requires understanding workload behavior, as prior studies of serverless^[30], microservice^[9], and batch^[32] workloads have shown」。这是全文方法论的总纲——**先测量、后设计**——而它的合法性来自三篇经典：

- **Serverless in the Wild**^[30] (ATC 2020, 微软 Azure Functions 团队)：用生产 trace 刻画了函数计算的调用模式与资源特征，发现函数冷启动与内存占用的分布规律，并据此改进了 Azure 的资源管理。它证明了「对新兴云负载做大规模表征」这个体裁本身的价值，也为本文提供了对照组数据：serverless 内存曲线近乎平坦 ($\text{peak/avg} \approx 1.5\times$) ——正是这个「平」，反衬出 agent 负载 $15.4\times$ 峰均比的「陡」 (§ 4.1)。
- **Resource Central**^[9] (SOSP 2017, 微软 Azure VM 团队)：系统刻画并预测了 Azure 虚拟机的资源使用，发现 VM 内存峰均比在 $2\text{--}3\times$ 区间，并用预测结果指导云资源调度。它为本文提供了「微服务/VM 负载」这个中间档对照，也是「预测式资源管理」路线的代表——恰好是本文 § 4.3 适应性错配要批判的范式。
- **Borg**^[32] (EuroSys 2015, Google)：Google 集群管理系统的经典论文，其资源回收与超卖策略全部建立在对批处理负载长期利用率的统计之上。本文用它代表「批处理/HPC」这一类确定性负载：执行时间以分钟到小时计、内存随数据规模平稳上升、峰均比约 $1\times$ (表 1)。

三条引用合在一起的逻辑是：本文不是发明「负载表征」这个新体裁，而是把它第一次应用到「AI agent 沙箱内部」这个从未被系统测量过的对象上，并且刻意沿用三剑客的对照框架（执行时长、镜像大小、内存峰均比、确定性）——表 1 的每一列维度都能在 [30][9][32] 里找到出处。这是学术论文最常见的继承方式：**方法继承 + 对象创新 + 定量对照**。

【零基础知识补给站】

如何用参考文献给论文「定位学术谱系」？ 一个实用技巧：不看论文引了什么「结论」，而看它在哪个句子里引。出现在「正如 X 所示，我们先测量再设计」里的引用，是方法论祖先；出现在「与 X 不同，我们的负载……」里的引用，是对照靶子；出现在「我们基于 X 构建」里的引用，是技术地基。一篇系统论文通常只需要三类祖先：体裁的祖先（本文是 [30][9][32]）、问题的祖先（本文是 agent 系统诸文）、工具的祖先（本文是 [14][15][23][41]）。把每条引用归到这三类之一，论文的谱系图就自动浮现了。

Insight#1 AgentCgroup 是「负载表征 → 系统设计」这一经典系统研究范式在 AI agent 时代的直系后代。

Insight#2 [30][9][32] 不只是致敬，更是定量对照的数据源：agent 的 $15.4\times$ 峰均比，是在它们 $1\times-3\times$ 的坐标系里才显得惊人。

Insight#3 方法继承 + 对象创新 + 定量对照，是低风险高可信的论文结构——本文是其教科书式范例。

15.2 Agent 系统研究新浪潮：问题域的同伴们

2024–2026 年，「为 LLM agent 造系统」成为系统研究的显学，论文引了整整一个方阵。按子方向再分四簇：

- **OS 抽象簇：AIOS^[25] 与 Trusted AI Agents^[6]**。AIOS (arXiv 2024) 提出「LLM agent 操作系统」的宏大概念，把 LLM 类比为 CPU、agent 类比为进程，主张在内核之上加一层 agent 调度与上下文管理层；[6] (arXiv 2025) 研究云端可信 agent 的执行环境。本文在 §1–§2 引它们，用以说明「OS 级抽象」方向已有呼声，但「无人对沙箱内的 OS 资源动态做过系统表征，也无人据此设计内核级控制」——两句话划出本文的空档。

- **观测与安全簇：AgentSight^[39]**。AgentSight (arXiv 2025，本文一作的同期工作) 用 eBPF 做 agent 的系统级行为观测与安全边界。本文 §5 明说原型「extending AgentSight」——仓库中 process 监控组件即派生自此（见第 13 章）。这条引用是**技术血缘**而非背景：观测层用 eBPF，控制层自然也用 eBPF。

- **调度与服务簇：Continuum^[22]、Astraea^[27]、Kairos^[7]、Nalar^[21]**。四篇都是 2025–2026 的 agent 调度系统：Continuum 管多轮对话的 KV cache 生命周期，Astraea 做状态感知调度，Kairos 解公共云高负载下的多 agent 低延迟服务，Nalar 是 agent 服务框架。它们的共同视角是「调度 LLM 推理与上下文」；本文的差异化视角是「调度沙箱内的 OS 资源」——同一问题域、正交的资源层。

- **成本与硬件簇：推理成本分析^[17]、异构系统^[4]、Kgent^[40]**。[17] (arXiv 2025) 从 AI 基础设施视角拆解 agent 的推理成本与 test-time scaling；[4] (arXiv 2025) 讨论用异构硬件高效服务 agent；Kgent (eBPF'24 workshop，本文一作更早的工作) 探索让 LLM agent 直接写内核扩展。Kgent 与本文的关系尤其值得玩味（笔者评注）：Kgent 让 agent 进内核写代码，AgentCgroup 让 agent 通过意图协议影响内核策略——同一作者对「agent × 内核」关系思考的两次迭代。

【零基础知识补给站】

怎么读「同一个作者引用自己上一篇论文」？学术圈的「自我引用」不全是刷指标，常常标记着一条研究线的演化轨迹。本文作者序里的 Yusheng Zheng、Andi Quinn 同时出现在 Kgent^[40] (2024) 与 AgentSight^[39] (2025) 的作者栏。三篇连起来读：Kgent 探索「agent 能写内核代码吗」，AgentSight 探索「能用内核技术看住 agent 吗」，AgentCgroup 探索「能用内核技术管好 agent 的资源吗」——一条清晰的「agent × OS」研究主线。顺着作者自引向前翻，是快速理解一个研究组思路的最短路径。

Insight#1 这十篇文献共同宣告「agent 系统研究」已成为独立方向；AgentCgroup 的位置是其中唯一的「沙箱内 OS 资源」视角。

Insight#2 AgentSight[39] 是技术血缘，Kgent[40] 是思想前传——自引勾勒出作者研究线的三年演化。

Insight#3 调度簇四篇管「LLM 侧」，本文管「工具侧」，正合 § 3 的发现：工具执行 + 初始化占了 55–60% 的时间，两边都是真问题。

15.3 Agent 框架与基准：实验对象的家谱

这一组文献定义了论文的**实验材料**：用什么 agent、在什么任务集上测、按什么范式交互。

- **ReAct**^[38] (ICLR 2023)：提出「推理—行动」(reason-then-act) 循环，让 LLM 交替产出思考与工具调用。它是所有 coding agent 的交互范式源头；论文 § 1 开篇即引，因为「reason-then-act 循环」正是「LLM 思考期资源平稳、工具调用期资源突发」这一两层结构 (§ 3.3) 的成因。
- **OpenHands**^[34] 与 **SWE-agent**^[37] (ICLR 2025 / NeurIPS 2024)：两个最有代表性的开源 coding agent 框架。本文没有直接用它们做实验，而是引它们论证「bash -c 拦截」的普适性——所有这类框架都把工具调用落成子进程命令。
- **SWE-bench**^[16] 与 **SWE-rebench**^[5] (ICLR 2024 Oral / arXiv 2025)：SWE-bench 用真实 GitHub issue 评测 agent 修 bug 的能力，是 coding agent 的事实标准基准；SWE-rebench 是 Nebius 团队的自动化、防污染版本，持续从新 issue 采集任务。本文的全部 144 个实验任务来自 SWE-rebench^[5]，其官方容器镜像 (2.9–17.3 GB) 也是 § 3 镜像大小数据的来源。
- **Agent 综述**^[33] (Frontiers of Computer Science 2024)：LLM 自主 agent 的大型综述，论文用它支撑「软件工程是 agent 代表性应用域」这一背景陈述。

这一组的引用逻辑是**可复现性背书**（笔者评注）：框架选公认开源的、基准选社区标准的、交互范式引原始出处——读者若想复现 § 3，每一步都有公开锚点。

【零基础知识补给站】

什么是基准 (benchmark)？为什么基准的选择影响论文可信度？ 基准是一套公认的任务集 + 评测协议，让不同研究的结果可比较。选基准有两个陷阱：一是「刷题污染」——模型训练时见过题目，SWE-rebench^[5] 用持续更新的新 issue 解决它；二是「基准与论点不匹配」——若本文要论证资源控制对**生产级** agent 有用，却只在玩具任务上测，结论就站不住。SWE-bench 系任务（真实仓库、真实 issue、完整测试套件）在规模与资源特征上最接近生产 coding agent，这是本文选择它并值得肯定的地方。

Insight#1 ReAct[38] 是理解本文资源曲线的钥匙：两层内存结构（基线 + 突发）正是 reason-then-act 循环在资源层的投影。

Insight#2 SWE-rebench[5] 的选择同时解决了污染问题与真实性问题，是本文表征结论可信度的地基之一。

Insight#3 框架引用（[34][37]）服务于机制论证而非实验——它们保证 bash 拦截点对全生态成立。

15.4 内核与云基础设施：技术地基与被批判的现状

数量最大的一组，可再分两簇：一簇是本文**站在上面**的内核机制，一簇是本文**批判**的现有方案。

地基簇：cgroup v2 官方文档^[14]（统一层级控制组，全文所有资源域的地基）；sched_ext 文档^[15]（CPU 侧机制，12.1 节）；eBPF verifier 文档^[23]（安全加载的保证，12.4 节）；libbpf 文档^[24]（原型实现所用的加载库与 CO-RE 工具链，§ 5）；memcg_bpf_ops RFC 补丁^[41]（内存侧机制，12.2 节——注意它是 41 条里唯一的「评审中的补丁」而非已确立的技术，呼应 12.3 节的风险讨论）。

靶子簇：PSI^[35]（压力信号标准接口）与基于它的两个用户态 OOM 管理器——Meta 的 oomd^[26] 和 systemd-oomd^[31]，三者共同构成 § 4.2 「响应性错配」的批判对象：信号到决策数十毫秒、容器级聚合无法归因到工具调用。TMO^[36]（ASPLOS 2022，Meta 的透明内存卸载系统）代表「渐进式内存压力换冷数据」路线，同样假设压力是渐进的、可换出的。K8s 三件套——QoS 分级^[19]、VPA^[20]、in-place resize^[18]——分别出现在 § 4.1 与 § 4.3：QoS 被批「Guaranteed 浪费、BestEffort 危险、Burstable 粒度仍错」，VPA 与 in-place resize 被批「分钟级调整对秒级突发无效」。Autopilot^[29]（EuroSys 2020，Google 基于历史 P95 的自动容量推荐）则是 § 4.3 适应性错配的头号靶子：历史分位数对非确定性负载在统计上失灵。

这一组的引用逻辑是「逐一点名、逐一证伪」：表 2 的三列（Static Limits / Reactive Control / Predictive Scaling）每一列都由具体文献支撑，批判因此不是稻草人论证。这种「把现有方案分三类、每类找到代表文献、指出共同假设失效」的写法，是 mismatch analysis 一节之所以有说服力的结构原因（笔者评注）。

【零基础知识补给站】

PSI、oomd、VPA 这些名词怎么快速建立感觉？ 可以按「反应速度」排成一列：K8s in-place resize^[18] 以分钟计 → VPA^[20] 以 Pod 重启/分钟计 → oomd^[26]/systemd-oomd^[31]（读 PSI^[35]）以数十毫秒计 → 本文的 eBPF 钩子以微秒计。AgentCgroup 的全部论点可以浓缩成一句话：**agent 的资源突发落在「微秒到 2 秒」这个现有工具全部够不着的时区里**。下次读到任何资源控制方案，先问它的反应时间常数，再看它与负载突发尺度的比值——这个比值基本决定了方案的生死。

Insight#1 地基簇五条全是内核官方文档级引用，唯独 memcg_bpf_ops[41] 是 RFC 补丁——论文技术风险集中在这一个引用上。

Insight#2 靶子簇的批判之所以有力，在于每条批判都对应 § 3 的一个实测数字（1–2 秒突发、3GB/s 变化率、15.4× 峰均比）。

Insight#3 表 2 的「三列三错配」结构可作为分析任何「新技术 vs 旧工具」问题的通用模板。

15.5 商业产品文档：问题真实性的背书

最后一组不是学术论文，而是 9 条厂商文档与博客：Claude Code 文档^[1]、Claude agent 安全部署指南^[2]、Apple Xcode agentic coding 发布稿^[3]、Devin 发布博客^[8]、Cursor 最佳实践^[10]、Claude Code 沙箱工程博客^[11]、GitHub Copilot coding agent 文档^[12]、Google Jules 发布稿^[13]、OpenAI Codex 发布稿^[28]。

它们在论文中的功能高度集中：全部出现在 § 1–§ 2，论证两件事。其一，**问题真实且正在爆发**——OpenAI、GitHub、Google、Apple、Cognition、Anthropic 全部在把 coding agent 做成商业产品，多租户托管的资源压力不是假想。其二，**沙箱是行业共识架构**——[2] 与 [11] 两篇 Anthropic 工程文档详述了 Claude Code 如何在沙箱中执行工具，为本文「所有实验与机制都发生在沙箱容器内」的设定提供了产业界注脚。

引用产品文档而非论文来支撑动机，在系统研究里是常见且正当的做法（笔者评注）：产业的采用速度经常跑在学术表征前面，「厂商都在做了、却没人量过它的资源行为」本身就是最有力的研究空白陈述。AgentCgroup 的 41 条文献里留出近四分之一给产品文档，传递的信号很明确：这篇论文站在产业与学术的接缝上，自觉地为前者补测量、为后者补问题。

【零基础知识补给站】

为什么学术论文会引用博客和产品文档？学术引用规范的核心不是「文献必须有同行评审」，而是「论断必须有可核查的出处」。当论断是「OpenAI 提供了这样的产品」时，最好的出处就是 OpenAI 的发布稿本身。阅读时只需注意分工：产品文档支撑「世界是什么样」的事实性论断，学术论文支撑「机制为什么有效」的因果性论断。本文 9 条产品引用全部用于前者，边界清晰。另外注意每条都标注了访问日期（2026-02-09）——网络内容会变，标注访问时间是引用网页资源的标准礼仪。

Insight#1 9 条产品文档的作用是「问题背书」：动机部分的可信度由产业界的实际采用来担保。

Insight#2 产品文档管事实、学术论文管因果——引用分工的清晰度是论文严谨性的一个侧面。

Insight#3 近四分之一的引用给产业界，呼应了本文的本质：为一门跑在学术测量前面的产业实践补上系统研究的功课。

第 16 章 批判性思考与展望

前十五章把论文「说了什么、做了什么、证据如何」拆了个遍。最后一章正文退后一步，做三件事：盘点局限（作者自述的与笔者补充的）、把这项工作放回 OS 演化的大图景里看它的意义、以及给不同身份的读者一份可带走的清单。

16.1 论文的局限

作者在 § 7 的局限陈述相当坦诚，可以归纳为三条：**表征覆盖窄**——只有一个 agent 框架（Claude Code^[1]）和一个基准（SWE-rebench^[5]），换框架换任务后四大发现是否成立未经验证；**评估是初步的**——trace replay + proof-of-concept，没有活的 LLM、没有生产规模；**机制覆盖不全**——原型只管 CPU 和内存，「初始化开销、大容器镜像、重试引起的内存累积」明确留作开放问题。

在作者清单之外，笔者依据仓库阅读补充四点。其一，**初始化占 31-48% 的大头未被机制触及**：AgentCgroup 优化的是「运行时」资源，但 § 3 的最大数字是 Podman 启动时的 user-namespace ID 重映射与镜像层开销——镜像均值 3.5 GB、范围 2.9-17.3 GB。沙箱启动优化（快照、预热、镜像瘦身）在收益空间上可能大于本文的全部机制之和。其二，**重试累积（最多 502 MB 未释放）与意图协议存在张力**：agent 若每次都「诚实」申报下一轮重试的内存，申报值会随累积漂移，协议没有考虑这种有状态的校准。其三，**内存侧机制依赖评审中的内核补丁**（12.3 节），部署时间线不受作者控制。其四，**评估场景是临界构造**（14.4 节）：29% 的 P95 收益是在 1100/1300 MB 精心设限的场景取得的，宽松内存下机制收益趋近于零——这不削弱论文，但削弱了任何「部署即省 29%」的朴素外推。

【零基础知识补给站】

如何批判性阅读一篇系统论文？ 一个实用的心法是**区分两类结论的稳健性**。「表征结论」（characterization）——负载长什么样——通常很稳健：144 个任务、两个模型测出来的「内存峰均比 15.4×」「185 MB 基线」这类数字，换平台大概率仍在同一量级；「系统结论」（system）——我的机制有多好——通常是初步的：它依赖场景构造、对照选择和实现完成度。对本文的正确读法因此是：**把 § 3 当可靠的新知识吸收，把 § 6 当机制可行性的存在性证明看待**。再检查三样东西：对照组是否回答论文的核心主张、场景参数是否被调过、作者标注的 preliminary 覆盖哪些句子。

Insight#1 作者的局限清单是负责任的，但它「避重就轻」地绕开了最大的收益池：31-48% 的初始化开销。

Insight#2 重试内存累积与意图协议的张力，是「适应性」主张内部尚未被注意到的一个理论裂缝（笔者观点）。

Insight#3 批判性阅读的**黄金分割**：表征结论当知识用，系统结论当方向用，两者用不同的置信度存放。

16.2 更大的图景：Agent 工作负载正在倒逼 OS 进化

把镜头拉远，AgentCgroup 位于一条近二十年的 OS 概念演化线上（笔者梳理）：2007 年 cgroup 随进程容器进入内核时，它管理的是「不懂事的进程」——策略由管理员静态写好，进程对自身的资源命运没有任何发言权；2015 年 cgroup v2^[14] 统一层级，云原生时代把它变成容器资源的事实底座，但交互模式没变：人定策略、内核执行、进程服从。AgentCgroup 第一次在这条线上引入了第三个角色：**能听懂反馈、会调整行为的负载本身**。资源控制从「管理」变成了「协商」——AGENT_RESOURCE_HINT 是负载对系统说话，[Resource] stderr 反馈是系统对负载说话。

这个转变的深层原因是论文 § 4.3 反复强调的非确定性：当负载的行为在原理上不可预测时，「预测—分配」范式在数学上就输了，唯一的选择是把负载拉进控制回路，让最了解自己下一步意图的一方参与决策。有趣的是，这条路并非全新发明，而是向互联网设计的回归（笔者评注）：TCP 拥塞控制之所以成功，正是因为端系统（最了解自己要发多少的一方）参与了拥塞协商。「把语义信息从端点引入系统决策」——end-to-end 原则在资源管理上的复活。

如果这条演化线继续延伸，可以预见的下一站是：**OS 接口的「agent 可读化」**。今天的内核接口（exit code、errno、信号）是为程序员设计的压缩编码；AgentCgroup 的 [Resource] 文案提示了一种新接口层——内核事件的结构化自然语言导出。再往下，意图协议可能从环境变量升级为标准化的 capability 协商格式，sched_ext^[15] 与 memcg_bpf_ops^[41] 这类可编程内核点，则让「每个租户自带资源策略」从定制内核的特权变成常规操作。

【零基础知识补给站】

什么是「控制回路」？为什么把负载拉进回路是范式变化？ 控制回路 = 感知 → 决策 → 执行 → 再感知的循环。传统资源管理的回路里，感知端是内核计数器、决策端是管理员预设的规则，负载只是被执行的对象。AgentCgroup 的回路多了一条边：决策结果以自然语言回流给负载，负载的下一轮行为因此改变——负载从「被控对象」升格为「控制者之一」。控制论里这叫「利用被控系统的内部模型」：当被控对象比控制器更了解未来需求时，分一部分决策权给它，整个系统的效率上限就抬高了。

Insight#1 cgroup 二十年：从「管理不懂事的进程」到「与会对话的负载协商」

——AgentCgroup 是这条演化线上的第一个拐点信号。

Insight#2 不可预测的负载倒逼「协商式」资源管理，这与 TCP 把拥塞决策交给端系统是同一个思想原型（笔者观点）。

Insight#3 「内核事件的 agent 可读化」可能是这篇论文最被低估的长期影响——它预示着 OS 接口层的一次范式迁移。

16.3 对从业者的启示

如果你在建设 agent 云平台、沙箱基础设施或内部 coding agent 服务，不必等这篇论文的机制成熟，以下结论今天就能直接用（均来自 § 3 表征，稳健性最高）：

- **内存是第一约束，按内存做容量规划。** agent 平均 CPU 利用率只有 7.6%–13.2%，而峰值内存 2–4 GB——128 GB 内存的机器按峰值只能装 32–64 个并发实例，此时 CPU 利用率不足

36%。为 agent 负载买 CPU 是浪费，买内存才是投资。

- **「185 MB 基线 + 突发」模型可直接进容量规划公式。**框架基线约 185 MB (Haiku 183、GLM 188) 是稳定的地板，突发部分按峰均比 $15.4\times$ 上界预留。64 并发 ≈ 12 GB 基线 + 突发余量——这是论文给出的、可直接抄进电子表格的数字。

- **优雅降级必须是默认策略。**OOM 对 agent 是任务级失败（丢上下文、冷启动 31–48%、重跑不收敛），不是进程级故障。任何「超限即杀」的策略都在拿任务成功率换内存利用率；memory.high 节流、超配延迟、降精度运行都比杀好。

- **工具调用级观测是低成本高回报的第一步。**本文的 bash wrapper 模式（替换 `/usr/bin/bash + memory.peak` 记账，第 11 章）不需要任何补丁内核，一天就能部署，立刻获得 per-tool-call 的峰值内存视图——先有数据，再谈控制。

【零基础知识补给站】

为什么「先有观测，再谈控制」？这是系统工程的铁律：你无法优化你看不见的东西。本文自己的历史就是证据——作者先做 AgentSight^[39]（观测）才做 AgentCgroup（控制）。对实践者，观测层的回报是即时的：哪些工具吃内存、哪个客户的工作流最重、重试循环浪费多少，这些答案直接转化为定价、配额和 SLO 的制定依据。而控制层（尤其涉及补丁内核的控制层）应该等观测数据证明「确实有这个瓶颈」之后再投入。

Insight#1 论文对工业界最硬的贡献不是 AgentCgroup 系统，而是那组可复用的容量规划参数：185 MB、 $15.4\times$ 、2–4 GB。

Insight#2 「不杀 agent」应成为 agent 基础设施的一号运维原则——杀进程等于毁任务，这笔账在传统运维直觉里是不存在的。

Insight#3 wrapper 观测模式是本论文中「今天就可用的」部分；补丁内核部分是「值得关注路线图」的部分，两者不要混着规划。

16.4 未来方向

按作者 § 7 的清单与笔者在阅读中发现的空白，未来工作可以分成四组，按价值排序（笔者判断）：

- **活的 (live) 评估。**用真实 LLM 在环的方式跑 AgentCgroup，测量端到端任务成功率、token 成本与完成时间的变化。这是把 § 6 从「机制存在性证明」升级为「系统收益证明」的唯一路径，也是论文自己列的第一优先级。
- **多框架、多基准验证。**在 OpenHands^[34]、SWE-agent^[37] 等框架上重跑表征，检验四大发现与意图协议的框架无关性。bash -c 拦截的普适性需要实证而非推断。
- **初始化与重试的专项优化。**31–48% 的初始化开销（快照恢复、镜像层懒加载）与重试内存累积（最多 502 MB，跨重试的内存回收策略）是两个比运行时控制更大的收益池，作者在 § 7 已点名，但解决方案需要「沙箱运行时与编排栈的协同优化」，难度高于本文机制。
- **意图协议的闭环学习。**让 agent 真正「学会」申报：把下行反馈的效果量化（收到 [Resource] 后，下一次申报的命中率是否提高？），并补齐 throttled 反馈与 cgroup.freeze 降级这两个论文已设计、仓库未实现的环节（13.4 节）。这是「意图驱动」从设计口号变成科

学结论的关键一跃。

【零基础知识补给站】

怎么判断一篇论文的 future work 哪些会真发生？ 看三点：依赖是否自主可控（live 评估只需要工程投入，最可能先发生；上游补丁合入依赖社区，时间不可控）、收益是否可量化（初始化优化有 31-48% 的现成数字做靶子，最容易立项）、以及是否有人竞争（多框架验证是学术快棋，抢发价值高）。用这个框架看本文：live 评估与多框架验证大概率一两年内出现；memcg_bpf_ops 的主线化^[41]是整条产品化路径上最关键的「他人之事」，值得持续关注 LKML。

- Insight#1** live 评估是本文到「可信系统」之间唯一绕不开的桥，其余方向都是扩建。
- Insight#2** 初始化与重试优化的收益池（31-48%）大于本文机制已证明的收益——沙箱运行时层可能是下一个被表征研究改造的领域。
- Insight#3** 「agent 能否在反馈中学会申报」是意图范式的核心科学问题；它被验证之前，「intent-driven」的 title 始终带一个星号。

第 17 章 附录

17.1 术语表

下表汇总本报告涉及的全部术语。三列分别为：术语（英文原词）、中文、一句话解释。按主题分组排列，方便回查。

术语	中文	一句话解释
Agent (AI agent)	智能体	由 LLM 驱动、能自主调用工具完成任务的软件实体。
Tool call	工具调用	agent 在推理循环中发起的一次外部命令执行（如跑 pytest）。
Reason-then-act loop	推理-行动循环	ReAct 范式：LLM 交替进行思考与工具调用 ^[38] 。
cgroup	控制组	Linux 内核按组对进程做资源核算与限制的机制。
cgroup v2	控制组 v2	2015 年设计的统一层级版本，一棵树管所有资源 ^[14] 。
memory.high	内存软限	超限触发回收与节流但进程不被杀的水位线。
memory.max	内存硬限	超限直接触发 OOM killer 杀进程的水位线。
memory.low	内存保护线	低于此线的 cgroup 在回收时受保护。
memory.peak	内存峰值	cgroup 历史最高内存占用，内核自动记账。
cgroup.procs	进程名单	写入 PID 即把进程迁入该 cgroup 的接口文件。
OOM killer	内存耗尽杀手	内核在内存耗尽时按评分选杀进程的最后手段。
eBPF	扩展伯克利包过滤器	在内核中安全运行用户自定义程序的虚拟机技术。
verifier	验证器	加载前静态检查 eBPF 程序安全性的内核组件 ^[23] 。
BPF Maps	BPF 映射	eBPF 程序与用户态/自身状态共享数据的键值存储。
struct_ops	结构体操作挂接	把 BPF 函数注册为内核子系统回调的挂接方式。
CO-RE	一次编译到处运行	借助 BTF 让 BPF 程序跨内核版本二进制兼容的技术（libbpf ^[24] ）。
sched_ext	可扩展调度器	允许用 BPF 定义 CPU 调度策略、出错自动回退的框架 ^[15] 。
memcg_bpf_ops	内存控制组 BPF 钩子	评审中的内核补丁：给内存控制器加 struct_ops 回调 ^[41] 。
page fault	缺页	进程访问未分配物理页时触发的内核分配动作，内存压力先行指标。
PSI	压力停顿信息	内核导出的「进程因等资源停滞多久」统计接口 ^[35] 。
oomd / systemd-oomd	用户态 OOM 管理器	Meta / systemd 基于 PSI 的用户态内存管理守护进程 ^{[26][31]} 。
QoS (K8s)	服务质量分级	K8s 的 Guaranteed/Burstable/BestEffort 三档 Pod 分级 ^[19] 。

术语	中文	一句话解释
VPA	垂直 Pod 自动伸缩器	K8s 按历史用量调整 Pod 资源配额的组件 ^[20] 。
P50 / P95 / P99	分位延迟	排队后位于 50%/95%/99% 位置的延迟值。
CV	变异系数	标准差与均值之比，衡量离散程度（本文峰值内存 CV=147%）。
trace replay	轨迹回放	按真实采集的资源曲线重演负载、换取可重复性的评估方法。
Podman	容器运行时	无守护进程的容器引擎，与 Docker 兼容，本文实验所用。
SWE-bench	软件工程基准	用真实 GitHub issue 评测 agent 修 bug 能力的基准 ^[16] 。
SWE-rebench	SWE-bench 再采集版	自动化、防污染、持续更新的版本，本文任务来源 ^[5] 。

Insight#1 术语按「负载层 → 内核机制层 → 评估方法层」三组记忆：先懂负载，再懂机制，最后懂怎么测。

Insight#2 memory.high 与 memory.max 的一字之差（软限与硬限）是全篇最重要的术语区分。

17.2 完整参考文献

以下为论文全部 41 条参考文献，按原编号从 arXiv:2602.09345v3 原文转写（已修正文本提取产生的异常连字符），每条后以破折号附一句中文注。

[1] Anthropic. 2026. How Claude Code Works. Claude Code Documentation.
https://code.claude.com/docs/en/how-claude-code-works (accessed 2026-02-09). — 本文实验所用 agent 框架的官方文档。

[2] Anthropic. 2026. Securely Deploying AI Agents. Claude API Documentation (Agent SDK Guides).
https://platform.claude.com/docs/en/agent-sdk/secure-deployment (accessed 2026-02-09).
— 论证沙箱是 agent 部署的行业共识。

[3] Apple. 2026. Xcode 26.3 unlocks the power of agentic coding. Apple Newsroom.
https://www.apple.com/newsroom/2026/02/xcode-26-point-3-unlocks-the-power-of-agentic-coding/ (accessed 2026-02-09). — 产业界采用证据之一。

[4] Zain Asgar, Michelle Nguyen, and Sachin Katti. 2025. Efficient and Scalable Agentic AI with Heterogeneous Systems. arXiv preprint arXiv:2507.19635. — 异构硬件服务 agent 的系统研究。

[5] Ibragim Badertdinov, et al. 2025. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. arXiv preprint arXiv:2505.20411.
Dataset: https://huggingface.co/datasets/nebius/SWE-rebench. — 本文 144 个实验任务的来源基准。

[6] Teofil Bodea, et al. 2025. Trusted AI Agents in the Cloud. arXiv preprint arXiv:2512.05951. — 云端可信 agent 执行环境研究。

[7] Jinyuan Chen, Jiuchen Shi, Quan Chen, and Minyi Guo. 2025. Kairos: Low-latency Multi-Agent Serving with Shared LLMs and Excessive Loads in the Public Cloud. arXiv preprint arXiv:2508.06948. — 多 agent 低延迟服务系统。

[8] Cognition. 2024. Introducing Devin, the first AI software engineer. Cognition Blog.
https://cognition.ai/blog/introducing-devin (accessed 2026-02-09). — 首个商用 AI 软件工程师产

品发布稿。

- [9] Eli Cortez, et al. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In SOSP. ACM, 153-167. — 微服务/VM 负载测量三剑客之一。
- [10] Cursor Team. 2026. Best practices for coding with agents. Cursor Blog. <https://cursor.com/blog/agent-best-practices> (accessed 2026-02-09). — 商业 coding agent 产品实践。
- [11] David Dworken and Oliver Weller-Davies. 2025. Beyond Permission Prompts: Making Claude Code More Secure and Autonomous. Anthropic Engineering Blog. Published Oct 20, 2025. <https://www.anthropic.com/engineering/claude-code-sandboxing> (accessed 2026-02-09). — Claude Code 沙箱架构的工程细节。
- [12] GitHub. 2026. About GitHub Copilot coding agent. GitHub Documentation. <https://docs.github.com/en/copilot/concepts/agents/coding-agent/about-coding-agent> (accessed 2026-02-09). — 产业界采用证据之一。
- [13] Google. 2025. Jules, Google's asynchronous AI coding agent, is out of public beta. Google Blog. <https://blog.google/innovation-and-ai/models-and-research/google-labs/jules-now-available/> (accessed 2026-02-09). — 产业界采用证据之一。
- [14] Tejun Heo. 2015. Control Group v2. Linux Kernel Documentation. <https://docs.kernel.org/admin-guide/cgroup-v2.html>. — cgroup v2 官方文档，全文机制地基。
- [15] Tejun Heo, David Vernet, and Josh Don. 2023. Extensible Scheduler Class. Linux Kernel Documentation. <https://docs.kernel.org/scheduler/sched-ext.html>. — sched_ext 官方文档，CPU 侧机制依据。
- [16] Carlos E. Jimenez, et al. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In ICLR. Oral. arXiv:2310.06770. — coding agent 评测的事实标准基准。
- [17] Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. 2025. The Cost of Dynamic Reasoning: Demystifying AI Agents and Test-Time Scaling from an AI Infrastructure Perspective. arXiv preprint arXiv:2506.04301. — 从基础设施视角分析 agent 推理成本。
- [18] Kubernetes Community. 2024. In-Place Update of Pod Resources. Kubernetes Enhancement Proposals. <https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/1287-in-place-update-pod-resources>. — K8s 原地调容 (alpha)，响应性错配的靶子之一。
- [19] Kubernetes Community. 2024. Pod Quality of Service Classes. Kubernetes Documentation. <https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/>. — K8s QoS 分级，粒度错配的靶子。
- [20] Kubernetes Community. 2024. Vertical Pod Autoscaler. Kubernetes Documentation. <https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler>. — 历史驱动调容，适应性错配的靶子。
- [21] Marco Laju, et al. 2026. Nalar: An Agent Serving Framework. arXiv preprint arXiv:2601.05109. — agent 服务框架。
- [22] Hanchen Li, et al. 2025. Continuum: Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live. arXiv preprint arXiv:2511.02230. — 多轮 agent 调度与 KV cache 管理。
- [23] Linux Kernel Community. 2024. eBPF Verifier. Linux Kernel Documentation. <https://docs.kernel.org/bpf/verifier.html>. — eBPF 安全验证器文档。
- [24] Linux Kernel Community. 2024. libbpf Overview. Linux Kernel Documentation. https://docs.kernel.org/bpf/libbpf/libbpf_overview.html. — 原型实现所用的 BPF 加载库。

- [25] Kai Mei et al. 2024. AIOS: LLM Agent Operating System. arXiv preprint arXiv:2403.16971. — 「LLM agent 操作系统」概念的先驱。
- [26] Meta/Facebook. 2024. oomd: A Userspace Out-of-Memory Killer. GitHub. <https://github.com/facebookincubator/oomd>. — Meta 用户态 OOM 管理器。
- [27] Hongqiu Ni, et al. 2025. Astraea: A State-Aware Scheduling Engine for LLM-Powered Agents. arXiv preprint arXiv:2512.14142. — 状态感知的 agent 调度引擎。
- [28] OpenAI. 2025. Introducing Codex. OpenAI. <https://openai.com/index/introducing-codex/> (accessed 2026-02-09). — 产业界采用证据之一。
- [29] Krzysztof Rzadca, et al. 2020. Autopilot: Workload Autoscaling at Google Scale. In EuroSys. ACM. — Google 历史 P95 驱动自动容量推荐。
- [30] Mohammad Shahradeh, et al. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In USENIX ATC. USENIX, 205-218. — Serverless 负载测量经典，本文方法论祖先。
- [31] systemd Project. 2024. systemd-oomd.service - A Userspace Out-Of-Memory (OOM) Killer. systemd Documentation. <https://www.freedesktop.org/software/systemd/man/systemd-oomd.service.html>. — systemd 用户态 OOM 管理器。
- [32] Abhishek Verma, et al. 2015. Large-Scale Cluster Management at Google with Borg. In EuroSys. ACM. — Google 集群管理经典，批处理负载代表。
- [33] Lei Wang, et al. 2024. A Survey on Large Language Model Based Autonomous Agents. Frontiers of Computer Science 18, 6 (2024), 186345. doi:10.1007/s11704-024-40231-1. — LLM 自主 agent 大型综述。
- [34] Xingyao Wang, et al. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In ICLR. arXiv:2407.16741. — 代表性开源 coding agent 框架。
- [35] Johannes Weiner. 2018. PSI - Pressure Stall Information. Linux Kernel Documentation. <https://docs.kernel.org/accounting/psi.html>. — 内核压力停顿信息接口文档。
- [36] Johannes Weiner, et al. 2022. TMO: Transparent Memory Offloading in Datacenters. In ASPLOS '22. ACM, 609-621. doi:10.1145/3503222.3507731. — Meta 透明内存卸载系统。
- [37] John Yang, et al. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In NeurIPS. arXiv:2405.15793. — 代表性开源 coding agent 框架。
- [38] Shunyu Yao, et al. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In ICLR. — 推理-行动循环范式的原始论文。
- [39] Yusheng Zheng, Yanpeng Hu, Tong Yu, and Andi Quinn. 2025. AgentSight: System-Level Observability for AI Agents Using eBPF. arXiv preprint arXiv:2508.02736. — 本文原型所扩展的 eBPF 观测系统（同作者）。
- [40] Yusheng Zheng, Yiwei Yang, Maolin Chen, and Andrew Quinn. 2024. Kgent: Kernel Extensions Large Language Model Agent. In ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions (eBPF '24). ACM, 30-36. doi:10.1145/3672197.3673434. — 让 LLM agent 编写内核扩展的先驱工作（同作者）。
- [41] Hui Zhu. 2026. mm: memcontrol: Add BPF hooks for memory controller. LWN.net. RFC PATCH bpf-next v3 00/12, Jan 23 2026. <https://lwn.net/Articles/1055698/>. — 内存侧机制依赖的评审中内核补丁。

Insight#1 41 条引用按功能分五类：方法论祖先、问题域同伴、实验对象、技术地基与靶子、产业背书——见第 15 章。

Insight#2 [41] 是全文唯一「未确立」的引用，也是全部技术风险的集中点。

17.3 复现指南摘要

以下基于仓库 docs/REPRODUCING.md、agentcg/memcg/EXPERIMENT_REPORT.md 与源码阅读整理。分「表征复现」与「系统复现」两条路径，前者门槛低得多。

环境要求

- 表征部分 (§ 3)：Linux + cgroup v2 + Python 3.10+ + Docker/Podman；数据已在仓库 experiments/ 目录（221MB 原始数据），运行 `python analysis/characterization.py` 即可重新生成论文图。
- 系统部分 CPU 侧 (§ 5/§ 6 sched_ext)：内核 6.12+（sched_ext 已合入主线），`clang/gcc/make/libbpf-dev/libelf-dev`；`make` 后 `sudo ./tests/test_weight_fairness.sh` 验证权重公平性。
- 系统部分内存侧（memcg_bpf_ops）：**必须自编译内核**——拉取 bpf-next 树，git am 应用 agentcg/memcg/patches.mbox（RFC v6，12 个补丁共 5571 行），编译安装并重启。论文评估机为 Linux 6.19.0-rc5 + 该补丁集。

wrapper 部署与运行

```
# 容器内安装 wrapper (agentcg/README.md)
cp /usr/bin/bash /usr/bin/real-bash
cp bash_wrapper.sh /usr/bin/bash
chmod +x /usr/bin/bash
export AGENTCG_ROOT="/sys/fs/cgroup/agentcg/session_high"
```

```
# 意图协议示例
AGENT_RESOURCE_HINT="memory:low" bash -c "git status"
AGENT_RESOURCE_HINT="memory:high" bash -c "pytest tests/"
AGENT_RESOURCE_HINT="memory:2g" bash -c "python train.py"
```

部署后每次 `bash -c` 调用的峰值内存、耗时、退出码会追加到 `/tmp/agentcg_tools.jsonl`；无 root 环境可改用 `bash_wrapper_local.sh` 模拟。

实验回放 (§ 6 数字)

```
# 内存隔离三策略对比（OOM 存活率 / P95 延迟场景）
sudo ./run_isolation_comparison.sh [--total-mb 1024] [--speed 10] [--runs 3]
```

```
# 开销微基准（P50 +0.3% 的来源）
sudo ./run_overhead_experiment.sh
```

```
# 单任务工具调用级回放（50x 加速对应 SPEED_FACTOR=50）
python scripts/replay_trace.py experiments/all_images_haiku/dask__dask-11628/attempt_1 --speed 10.0
```

【注意】

两处文档漂移需注意（笔者依据仓库核实）：docs/REPRODUCING.md 引用的 scx_flatcg/、memcg/multi_tenant_test/bpf_loader/ 路径在当前布局下不存在，实际代码在 agentcg/scheduler/ 与 agentcg/memcg/，且无 bpf_loader/ 子目录；agentsight 与 paper-repo 两个 git 子模块未初始化（空目录），论文 LaTeX 源码不在本地仓库中。复现时以实际目录结构为准。

Insight#1 没有补丁内核时的最佳路径：表征复现 + wrapper 观测 + CgroupMemcgController 回退后端，三者都不需要自编译内核。

Insight#2 仓库 isolation_results/ 与 overhead_results/ 保存了 2026-02-08 多轮原始 JSON 与 dmesg，论文数字可逐一对账。

17.4 本报告说明

- **解读对象：**论文《AgentCgroup: Understanding and Controlling OS Resources of AI Agents》，arXiv:2602.09345v3 [cs.OS]，2026 年 7 月 22 日版；作者 Yusheng Zheng（UC Santa Cruz）、Jiakun Fan / Quanzhi Fu（Virginia Tech）、Yiwei Yang / Andi Quinn（UC Santa Cruz）、Wei Zhang（UConn）。
- **源码依据：**配套开源仓库 github.com/eunomia-bpf/agentcgroup 本地检出，HEAD 为 551ca16（Correct nested sub-agent timing analysis (#8)）；全部代码引用逐行核对自该检出。
- **图片说明：**图 1-8 为论文原图，逐张保留并标注「论文原图」；图 D1-D5 为本报告为零基础读者加绘的示意插图，与论文无关。
- **观点标注：**凡论文与仓库原文以外的判断，一律以「（笔者推断）」「（笔者评注）」或 Insight 形式标出，与原始材料严格分离。
- **数字原则：**所有测量数字均忠于论文原文或仓库 EXPERIMENT_REPORT，不做二次换算后替代原始值。

Insight#1 本报告的可信度建立在三重可核查性上：论文原文、仓库源码、实验原始数据，三者互相印证。

Insight#2 读罢全文若只记得一句话：agent 时代，操作系统第一次有了「能听懂反馈的负载」——控制论的回路里多了一个会说话的节点。