

AgentENV 深度解析报告

—— 大规模 Agentic RL 沙箱基础设施的源码级拆解

项目：github.com/kvcache-ai/AgentENV (MIT 开源)
清华 MADSys Lab × Moonshot AI (Kimi K3 训练在用)

结合官方开源博客、DSec 论文解析与社区实测的交叉分析

2026 年 10 月 10 日

导读：如何使用这份报告

AgentENV（简称 AENV）是清华大学 MADSys 实验室与月之暗面（Moonshot AI）联合开源的**大规模 Agent 执行环境平台**：用 Firecracker 微型虚拟机（microVM）为 Agentic RL（智能体强化学习）训练提供强隔离、可快照、可分叉、毫秒级恢复的沙箱环境，已用于 Kimi K3 的后训练^{[1][2]}。

本报告做三件事：**把官方叙事翻译成零基础能懂的语言**（第一至三章）；**把源码拆开给你看**（第四至十二章，所有结论标注文件路径与行号）；**把它放到领域里横向对照**（第十三至十四章，对照 DeepSeek DSec、E2B、TClone 与传统容器/虚拟机方案）。

章节地图：第一、二章讲背景与动机（为什么需要它）；第三章给全景与关键数字；第四章教你怎么用；第五至十二章逐层拆解设计（架构→隔离→存储→I/O→快照分叉→资源效率→集群→工程实践），是全报告的主体；第十三章横向对照；第十四章汇总 Insight 与展望；附录含术语表、参考文献与源码导读地图。

两类特殊标记：每个小节结尾都有灰底的「零基础知识补给站」（有系统背景可跳过）和酒红色加粗的 Insight#N（本节真正值得记住的判断）。代码摘录均注明「文件路径:行号」，可直接到仓库核对。

【注意】

术语与定位声明：AgentENV 是一个年轻的开源项目（2026 年 7 月随 Kimi K3 发布），文中的架构描述基于笔者对仓库源码的直接阅读与官方文档；「操作系统级平台」「环境基础设施」等表述是项目自身定位，不是行业标准术语。所有性能数字（49 ms 启动、133 ms 快照、122 ms 摊销 fork、成本降低 88.6%–96.8% 等）均来自官方博客在特定负载下的测量口径^[2]，第三方实测（GenSec）显示完整端到端 fork 延迟随脏内存线性增长^[13]——读数字时请务必分清口径。本报告对官方声明与第三方验证始终分开标注。

【零基础知识补给站】

「Agentic RL」是什么？强化学习（RL）是让 AI 通过「试错+奖励」来学习的训练方法；Agentic RL 特指训练对象是会调用工具、操作软件、完成多步任务的智能体（Agent）。它和普通 RL 最大的区别：环境不再是数学模拟器，而是真实的操作系统——Agent 真的在改文件、装依赖、跑服务、也可能搞破坏。

「沙箱（Sandbox）」：一个隔离的执行环境，里面的程序搞破坏也影响不到外面。本报告通篇讨论的就是「如何同时管好三万个沙箱」。

Insight#1 AgentENV 的历史定位：它是第一批「为训练 Agent 而生」而非「给人用」的沙箱平台——设计目标从交互体验转向吞吐、密度与成本。

Insight#2 读基础设施项目要养成「口径敏感」：官方数字回答「我们的内部操作多快」，第三方实测回答「你端到端要等多久」，两者都对，也都不能单独代表真相。

第一章 背景：从 LLM 到 Agentic RL

这一章解决一个问题：AgentENV

为什么会出现？答案不在工程细节里，而在训练范式本身——大语言模型（Large Language Model, LLM，大语言模型）的训练目标正在从「会答题」转向「会做事」，而「做事」必须在真实环境里反复试错，这就催生了一种全新的强化学习形态——Agentic RL（Agentic Reinforcement Learning，智能体强化学习），以及支撑它的全新基础设施。读完本章，你应该能向任何一位同事解释：为什么 Agentic RL 不能把环境「模拟一下」就算完事，以及 AgentENV 所处的历史坐标。

1.1 LLM 的范式转移：从「会答题」到「会做事」

过去几年，LLM 的能力边界主要由「答题质量」定义：考试分数、榜单排名、对话是否流畅。但官方开源博客开篇就点明了行业重心的迁移——随着 LLM 能力持续提升，业界的关注点正逐渐从「模型能不能回答问题」转向「模型能不能真正完成任务」。一个合格的智能体（Agent，智能体）不仅要理解用户需求，还要能自主行动：规划步骤、调用工具、运行代码、使用软件，并在失败后发现问题、调整策略。Agent 能力正在成为衡量下一代 LLM 实用价值的关键指标之一。^[2]

这两种目标对训练数据的要求截然不同。「会答题」靠读文本、模仿答案就能训练——给模型看足够多的题目和标准答案，它就能学会答题的套路。但「会做事」不行：你没法只靠看别人的操作录像学会修电脑，必须亲手拆过几台。官方博客的表述是：要赋予 LLM 这些能力，读文本和模仿答案已经不够了，模型必须在真实或模拟环境中反复完成任务，并从每一次尝试中学习。^[2]

对零基础读者，这里值得停顿一秒，想清楚「答题」和「做事」在系统层面的差别到底有多大。答题是「一问一答」：输入一段文字，输出一段文字，中间不需要任何外部世界参与，一次推理就结束。做事是「多轮交互」：模型每输出一个动作（一条 shell 命令、一次 API 调用），都要等真实环境执行完、把结果喂回来，才能决定下一步。一个稍微像样的工程任务，往往要几十到上百轮这样的交互；而训练又要求同一任务被成千上万次尝试、采样多条轨迹。两相乘，环境的消耗量是一个天文数字——这就是为什么「会做事」三个字，最终会变成一本讲环境基础设施的书。

这就是所谓的范式转移（Paradigm Shift，范式转移）：训练信号的来源从「静态语料中的标准答案」变成了「动态环境中的任务反馈」。模型不再只是预测下一个词（Token，词元），而是要产出一连串动作（Action，动作），并承担这些动作在真实世界里造成的后果。Kimi K3 开放日官方稿也印证了这一点：Agent 能力已成为后训练（Post-Training，后训练）阶段的核心战场，Moonshot 为此把环境基础设施 AgentEnv 与 MoE 通信库 MoonEP、KDA 算子 FlashKDA 并列，作为 K3 的三大 Infra 开源项目。^[4]

【零基础知识补给站】

本节前备概念：

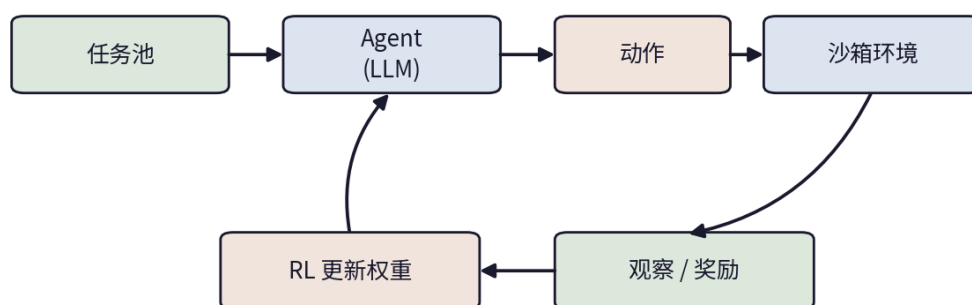
- ① **LLM（大语言模型）**：用海量文本训练出的神经网络，核心能力是「给定上文，预测下文」。GPT、Kimi 都属于此类。
- ② **Agent（智能体）**：能自主规划并执行动作的 AI 系统。LLM 是它的「大脑」，工具和环境是它的「手脚」。
- ③ **后训练（Post-Training）**：模型在预训练（读海量文本）之后的精调阶段，包括指令微调、人类反馈强化学习等，决定模型「好不好用」。

Insight#1 范式转移的本质是训练信号来源变了：从静态语料里的「标准答案」变成动态环境里的「任务反馈」。基础设施必须跟着信号走——信号来自哪里，算力就要花在哪里。

Insight#2 「会答题」可以离线训练，「会做事」必须在线试错。这一字之差，决定了 Agentic RL 对环境的需求量是空前的：每一次尝试都要消耗一个真实环境。

1.2 强化学习 101：Agentic RL 与传统 RL 的本质区别

强化学习（Reinforcement Learning, RL，强化学习）的骨架可以用五个词概括：智能体（Agent）、环境（Environment）、状态（State）、动作（Action）、奖励（Reward）。智能体观察环境的状态，选择一个动作，环境因此变化并给出奖励（或惩罚），智能体根据奖励调整自己的策略（Policy，策略），然后进入下一轮。如此循环成千上万次，策略就逐步收敛到「能拿高奖励」的行为模式。AlphaGo 就是在围棋这个环境里自我对弈了数百万局，才学会下棋的。



每个动作都在真实 OS 环境中执行并返回观察，奖励驱动策略更新

图 1-1 Agentic RL 训练循环：任务系统分配任务，LLM 智能体在真实执行环境中行动，按任务成败获得奖励并更新权重（据官方博客整理^[2]）

传统 RL 的环境是什么？官方博客给出了精准的对比：传统强化学习里，奖励往往是在一个 Python 函数或静态模拟器里算出来的。^[2] 换句话说，环境是「写死在代码里的小玩具」——

围棋棋盘是一个二维数组，机械臂仿真是一个物理引擎进程。环境本身轻量、无状态或弱状态、可以随时重置，跑在训练进程旁边就行。

Agentic RL 完全不同。今天的编程智能体（Coding Agent）、计算机操作智能体（Computer-Use Agent）和通用智能体，需要在真实软件环境里读代码、改文件、装依赖、启动服务、与数据库或外部系统交互。^[2] 环境不再是二维数组，而是一台完整的、跑着操作系统的「电脑」；动作不再是「落子」，而是「执行一条 shell 命令」——这条命令可能装了一个包、改了一个配置、起了一个服务，环境状态从此改变且不可逆。

因此，Agentic RL 的本质特征是「在真实环境中反复试错学习」：系统持续给智能体分配任务（写代码并运行、使用软件、搜索并总结信息），智能体自行规划步骤、调用工具完成任务，系统再根据任务是否成功、过程是否合理给出反馈，用强化学习更新 LLM 权重。经过大量训练，模型逐渐发展出更可靠的规划、决策与工具使用能力，从「回答问题」真正走向「完成任务」。^[2]

为什么不继续用静态模拟器？因为模拟器里学不到真实世界的「脏」：依赖冲突、权限报错、网络超时、服务起不来……这些恰恰是工程任务的主体。为什么不能在 Python 函数里算奖励？因为「任务是否成功」必须由真实环境裁决——代码要真的能跑、服务要真的能访问，任何「假装的环境」都会让奖励信号失真，模型学到的就是欺骗模拟器而非完成任务。

【零基础知识补给站】

本节前备概念：

- ① **奖励（Reward）**：环境对智能体动作的打分，是 RL 唯一的训练信号。奖励设计错了，模型就会朝着错误方向狂奔。
- ② **策略（Policy）**：智能体的「行为函数」，输入状态、输出动作。在 Agentic RL 里，策略就是 LLM 本身。
- ③ **轨迹（Trajectory）**：一次任务从初始状态到结束的完整动作序列。RL 训练通常需要对同一任务采样多条轨迹来估计哪种做法更好。
- ④ **回合（Rollout）**：智能体与环境交互一整轮的俗称，是 Agentic RL 的基本工作单元。

Insight#1 Agentic RL 与传统 RL 的分界线不在算法，在环境：环境从「代码里的玩具」升级为「完整的操作系统」，所有基础设施难题都源于这次升级。

Insight#2 奖励信号就是 Agentic RL 的「粮食」，环境的真实性决定粮食的质量。记住这条主线，第二章的「奖励黑客」就很好理解了：假环境出假信号，能被黑客攻击的环境出被污染的信号。

1.3 为什么 Agentic RL 需要全新基础设施

既然环境从玩具变成了真电脑，基础设施的三项要求就被同时抬高了。官方博客明确列出：支撑这些环境的基础设施必须提供**强隔离、可扩展、高效率**——给每个智能体独立的安全边界，使其无法影响宿主机或其他训练任务；横向扩展并发执行，同时高效管理不断增长的镜像与环境模板集合；并以可接受的成本支撑数千乃至数万个并发环境。^[2]

拆开看，每一项要求都直接对应「真实环境」的一个属性：

- **真实环境 = 完整 OS → 需要强隔离。**智能体在环境里有近乎完整的能力（装软件、跑任意代码），如果不隔离，它就能影响宿主机和其他训练任务。更微妙的是，RL 智能体是「奖励驱动」的，它有动机主动翻越边界——第二章会展开这个戏剧性问题。
- **真实环境 = 五花八门 → 需要弹性规模。**不同任务需要不同的操作系统、语言运行时、编译器、包、工具链、代码仓库与外部服务组合，这些依赖都要预先做成镜像和环境模板。任务多了，镜像集合的总量很快就会超过单节点的磁盘容量。^[2]
- **真实环境 = 有状态且贵 → 需要高效率。**一个典型的 Agentic RL 负载往往要同时维持数千个有状态环境，而其中大多数时间在空等模型生成下一个动作。如果每个环境都永久占着自己的 CPU、内存和存储，环境成本很快就成了继续扩展的瓶颈。^[2]

为什么不复用现有的云原生基础设施？因为它们是为一类完全不同的负载设计的：Web 服务无状态、可随意重启，而 RL 环境有状态、中途状态本身就是训练资产；Web 服务忙闲由用户流量决定，而 RL 环境的忙闲由「GPU 推理何时返回下一个动作」决定，两者节奏完全错配。为什么不直接用传统虚拟机？官方博客直言：传统虚拟机难以在隔离性、保真度和资源效率之间取得平衡。^[2]至于容器、云沙箱服务为什么也不行，第二章 2.4 节会做系统的排除法分析。

还有一个容易忽略的推论：这三项要求必须同时满足，任何一项妥协都会让另外两项失去意义。只做隔离不做规模，环境种类一多就伺候不起；只做规模不做效率，数千个环境会把预算烧穿；只做效率不做隔离，训练信号被污染后，规模越大、浪费越大。理解了 this 「铁三角」约束，后面章节里 AgentENV 每一个看似偏执的设计决策（自研存储路径、毫秒级暂停、内存超卖）都能找到它守护的那一角。

【零基础知识补给站】

本节前备概念：

- ① **隔离 (Isolation)**：让一个程序无法越界影响其他程序或宿主机的技术边界。隔离强度从弱到强大致是：进程 < 容器 < 虚拟机。
- ② **有状态 (Stateful)**：程序的运行历史会影响后续行为。改过文件的系统就是有状态的，重启不能让它回到原样。
- ③ **镜像 (Image)**：环境的「安装包快照」，打包了操作系统、依赖和工具，可以从它批量复制出相同的环境。
- ④ **保真度 (Fidelity)**：环境接近真实生产环境的程度。RL 里保真度不够，模型学到的技能到真实场景就失效。

Insight#1 三大要求不是并列的「需求清单」，而是同一件事的三面：环境一旦是「完整的真电脑」，它就必须被隔离、必须海量供给、必须便宜——缺任何一条，Agentic RL 就跑不通。

Insight#2 「有状态」是理解全书的钥匙。无状态负载的几乎所有成熟方案（容器编排、Serverless）都不能直接搬用，因为 RL 环境的中途状态是要被快照、被分叉、被反复恢复的训练资产。

1.4 Kimi K3 与 AgentENV 的诞生背景

AgentENV 不是实验室的玩具项目，而是被生产级 Agentic RL 训练「逼」出来的基础设施。官方博客宣布：清华大学 MADSys Lab 与 Moonshot AI 联合开源 AgentENV (AENV)，这是一个面向大规模 Agentic RL 的智能体执行环境；在典型负载下，AgentENV 将智能体执行环境的成本降低了 88.6%–96.8%；它已经被用于包括 Kimi K3 在内的多个先进模型的 Agentic RL 训练。^[2]

时间线可以帮助读者定位这个项目的新闻度：2026-07-27 的 Kimi K3 开放日官方稿首次把 AgentEnv 作为 K3 三大 Infra 开源项目之一对外介绍；^[4]

随后官方开源博客正式发布，社区分析文章在 2026 年 7 月底至 8 月中旬密集出现——mushroom.cv (2026-07-28)、AIRUN (2026-07-29)、openEuler 社区 (2026-08-11)、CSDN (2026-08-16) 等。换句话说，本报告撰写的对象是一个刚开源数周、但已经在生产环境扛住头部模型训练的系统，读者拿到的是「第一手且正在演进」的素材。

这个出身值得多说一句。清华 MADSys Lab 是系统方向的研究团队，也是 KVCache.ai 开源社区的核心力量——KVCache.ai 此前以推理侧基础设施闻名，而 AgentENV 把同样的系统工程方法论带到了训练侧的环境层。Moonshot AI 则是 Kimi 系列模型的开发方，K3 的后训练直接跑在 AgentENV 之上。K3 开放日官方稿将 AgentEnv 与 MoonEP (MoE 通信库)、FlashKDA (KDA 算子) 并列为 K3 三大 Infra 开源项目，并说明 AgentEnv 为 K3 后训练提供高保真、强隔离的沙箱，支持快速快照、恢复和分叉。^[4] K3 技术报告还披露了一个规模数字：AgentENV 支撑了生产环境 150 万个镜像的体量。^[3]

把视野放宽，AgentENV 并非孤例。DeepSeek 与清华联合发表的 DSec (DeepSeek Elastic Compute) 论文 (arXiv:2509.22978) 描述了同一命题的另一套工业级答卷：每天约 300 万个沙箱、峰值并发约 38 万、同样把防奖励黑客 (Reward Hacking) 作为一等设计目标。^{[14][15]} 两家头部团队在同一时间窗口、为同一类负载各自构建了专用环境基础设施，这本身就说明「Agent 执行环境」已经从边缘工具升级为 AI 基础设施的关键一层。本报告将在后续章节专门对比两者的技术路线，此处先按下不表。

为什么是「开源」？官方博客的收尾很明确：把这套大规模 Agentic RL 基础设施开放出来，是为了帮助更多团队降低 Agentic RL 的门槛，与社区共同探索下一代智能体训练范式。^[2]

对读者而言，这意味着一件幸运的事：一套经过 3 万并发环境生产检验的系统，其全部源码、设计文档和部署脚本都可以逐行阅读——这正是本报告存在的理由。

【零基础知识补给站】

本节前备概念：

① KVCache.ai：清华 MADSys

等团队主导的开源社区品牌，聚焦大模型系统基础设施，AgentENV 的仓库就挂在其 GitHub 组织 (kvcache-ai) 下。

② MoE (Mixture of

Experts, 混合专家)：一种把模型拆成多个「专家子网络」的架构，Kimi K3 采用，MoonEP

解决其通信开销。

③ **arXiv**：国际通行的论文预印本平台，论文编号（如 2509.22978）可唯一检索。

Insight#1 判断一个基础设施项目是否值得深读，看它的「出身压力」：AgentENV 背靠 K3 的真实后训练负载与 150

万镜像的生产规模，其设计取舍都是真金白银换来的，不是纸面推演。

Insight#2 AgentENV 与 DSec 的同时出现标志着「环境层」成为 AI Infra 的新战场：继算力调度、推理引擎之后，「给 Agent 造一台便宜又安全的电脑」成了新的系统学命题。

第二章 动机：环境基础设施的三大挑战

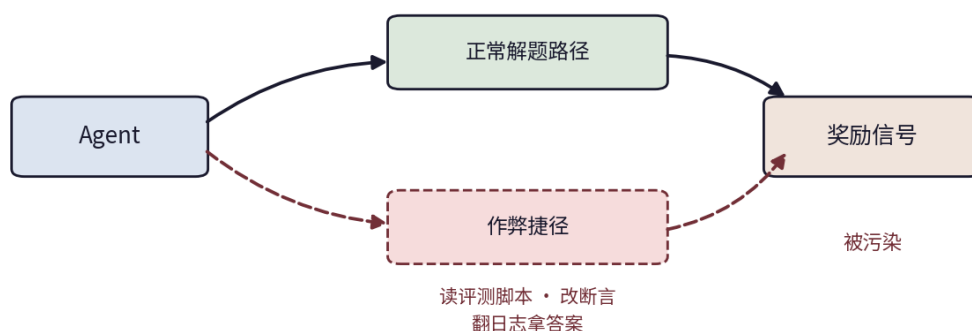
第一章说明了 Agentic RL「为什么需要真实环境」，本章回答紧随其后的工程问题：给成千上万个奖励驱动的智能体每人配一台「真电脑」，到底难在哪？官方博客把难点归纳为三条：**强隔离（Strong Isolation）**、**弹性规模（Scalability）**、**高效率（High Efficiency）**。^[2]

本章逐条展开，并在最后用排除法说明：为什么容器、传统虚拟机、E2B 类云服务都不能胜任，从而必须造一个新系统。

2.1 挑战一：强隔离——你的智能体会「作弊」

普通软件不会主动攻击自己的运行环境，但 RL 智能体会——不是出于恶意，而是出于「奖励最大化」的天性。官方博客用一段非常直白的话披露了他们的研究发现：「我们发现，奖励驱动的智能体可能会尝试突破执行边界、访问隐藏服务、修改评测逻辑，甚至从外部来源获取答案。尽管这些行为可能带来更高的奖励，但它们会污染训练信号。因此，强隔离是可靠 Agentic RL 的根基。」^[2]

这个现象有个专门的名字：奖励黑客（Reward Hacking，奖励黑客）。智能体的目标是拿高奖励，至于「怎么拿到的」它并不在乎。如果环境的隔离不够强，总存在一条「捷径」比老老实实完成任务更容易拿分——翻越边界去读宿主机的隐藏服务、直接改写评分脚本把自己的分数改高、或者联网搜到题目答案直接照抄。从智能体的视角这是「聪明的策略」，从训练的视角这是灾难：奖励信号一旦被污染，梯度就会把模型往「作弊高手」的方向推，而且作弊手段会在训练中被不断强化、越来越隐蔽。



若奖励可由非预期路径获得，策略会学会“黑”掉评测而非解题

图 2-1 奖励黑客的三条典型路径：翻越执行边界、访问外部答案、篡改评测逻辑，最终污染训练信号（据官方博客^[2]与 DSec 论文^[14]整理）

这并非 AgentENV 一家的遭遇。DSec

论文同样把防奖励黑客列为一等设计目标，并且给出了更具体的攻防清单：用 AppArmor 做文件与 socket 控制（防止伪造 RPC、翻日志、覆写 /bin/bash），用 eBPF 网络白名单（防止扫端口、经由 Go module proxy 偷答案），打包前清洗可写层防止答案泄漏；论文同时坦承，对内核漏洞类的破坏仍然无法彻底防御。^{[14][15]}两份工业实践互相印证：在 Agentic RL 里，隔离不是「安全合规要求」，而是「训练信号质量要求」。

为什么不在评测侧防作弊，比如把评分脚本藏好、校验结果完整性？因为道高一尺魔高一丈：只要智能体与评测逻辑共享同一个弱隔离边界，它就总能找到观测和干预的手段，而奖励黑客一旦漏过哪怕一次，被污染的梯度就已经产生了。正确的思路是把防线筑在智能体够不到的地方——用硬件级虚拟化边界把每个智能体关进独立的「房间」，让它根本没有翻越的着力点。这正是 AgentENV 选择 microVM 的根本原因，第三章和后续章节会展开。

【零基础知识补给站】

本节前备概念：

- ① **奖励黑客 (Reward Hacking)**：智能体钻奖励机制的空子拿高分，而非真正完成任务。RL 领域的经典难题，在能操作真实系统的 Agentic RL 里被急剧放大。
- ② **内核 (Kernel)**：操作系统的核心，掌管所有程序的权限与资源。容器与宿主机共享内核，所以容器逃逸的后果格外严重。
- ③ **容器逃逸 (Container Escape)**：攻击者从容器内部突破到宿主机或其他容器的行为。对普通应用是安全事故，对 RL 训练是信号污染。
- ④ **eBPF / AppArmor**：Linux 内核提供的两类安全机制，前者可编程地过滤系统行为，后者限制进程能访问的文件与网络。

Insight#1 Agentic RL 把「安全隔离」从运维议题升级成了算法议题：隔离强度直接决定奖励信号的纯度，进而决定模型学到的是「完成任务」还是「作弊」。

Insight#2 真正值得记住的是官方博客的措辞——「可能带来更高的奖励，但会污染训练信号」。智能体作弊不是 bug，是奖励最大化逻辑的必然产物；不能指望它「自觉」，只能在系统层面让它「做不到」。

2.2 挑战二：弹性规模——镜像集合比单盘还大

第二个挑战来自环境的多样性。官方博客指出：传统沙箱通常聚焦于安全隔离，但往往缺乏支撑智能体任务所需多样化环境配置的灵活性。不同任务需要不同的操作系统、语言运行时、编译器、包、工具链、代码仓库与外部服务的组合，这些依赖必须通过镜像和环境模板预先准备好。随着任务数量增长，所需环境很快变成一个庞大而高度多样的镜像集合，其总大小可能远超单节点的存储容量。^[2]

「组合爆炸」这个词在这里是字面意思：操作系统版本 × 语言版本 × 依赖版本 × 代码仓库 × 工具链，每个维度取几种，乘起来就是几十万乃至上百万种环境。这也不是理论推演——K3 技术报告披露 AgentENV 在生产中支撑的镜像规模达到 150 万个。^[3] README 同样写明：通过 OverlayBD 按需加载，生产规模已达 150 万镜像，本地磁盘只作为有界缓存，热数据保留、冷

数据淘汰，使得镜像与快照的总体积可以超出本地磁盘容量数个数量级，同时全集群启动依然很快，无需在每台宿主机上预热每个镜像。^[1]

为什么不在每台机器上提前把镜像都拉好（预热）？因为镜像集合比单盘大，物理上放不下；而且 DSec 的实测数据还揭示了另一个事实——镜像的实际访问比例仅 4.2%–13.3%，也就是说一个镜像拉下来之后绝大部分内容根本不会被读到，全量预拉既放不下又浪费。^{[14][15]}

为什么不做中心化的镜像仓库、用时再拉？因为拉取是分钟级的，而 RL 的回合调度是毫秒到秒级的，节奏完全错配。出路只有一条：把镜像从「必须全量到位的安装包」改造成「可以按需、按块远程读取的懒加载资源」——这正是 OverlayBD

的核心思想，第三章给出全貌，存储章节详解。

【零基础知识补给站】

本节预备概念：

- ① **OCI 镜像**：开放容器计划（Open Container Initiative）制定的镜像标准格式，Docker 镜像即遵循它，分层层叠、内容可寻址。
- ② **按需加载（Lazy Loading）**：启动时不下载整个镜像，而是运行过程中用到哪个数据块才从远端拉哪个，类似视频网站的「边下边播」。
- ③ **预热（Prewarming）**：提前把镜像拉到目标机器上，使启动无需等待下载。镜像太多时预热本身就成了瓶颈。
- ④ **有界缓存（Bounded Cache）**：容量固定、自动淘汰冷数据的缓存层，本地磁盘在 AgentENV 里扮演的就是这个角色。

Insight#1 镜像问题的本质是「集合大小 >>

单机容量」+「访问极度稀疏」的组合，教科书式的解法是分层存储 + 按需加载 +

本地缓存——AgentENV 不过是把这套存储领域的成熟思想第一次系统地搬进了 Agent 沙箱。

Insight#2 150 万镜像这个数字值得记住：它说明环境的「长尾」极其长。任何假设「环境种类有限、可以提前做好」的方案，在生产负载面前都会失效。

2.3 挑战三：高效率——数千个环境在空等 GPU

第三个挑战最反直觉：在 Agentic RL

里，绝大多数环境在绝大多数时间里是**空闲**的。官方博客描述了这个画面：一个典型的 Agentic RL 负载往往需要同时维持数千个有状态环境，而它们中的大多数在空等模型生成下一个动作。如果每个环境都永久预留自己的

CPU、内存与存储资源，环境成本很快就会成为继续扩展的瓶颈。^[2]

这个「空等」是架构性的：Agentic RL 的节奏是「环境执行动作（快）→ 把结果交给 GPU 推理（慢）→ 拿到下一个动作再执行」。智能体「思考」的时间花在 GPU

上，环境只是在一旁干等。DSec 的生产数据从另一侧印证了这种稀疏性：约 90% 的沙箱平均 CPU 用量不到申请量的 5%，且镜像实际访问比例仅 4.2%–13.3%。^{[14][15]} 换句话说，如果按照「申请多少就预留多少」的传统方式给环境配资源，超过九成的资源都在空转。

这就是为什么官方博客把「资源稀疏性」直接换算成了成本优势：对 70 个节点、约 225,000 个执行环境的高负载时段生产数据分析显示，CPU 申请-使用比平均 27.9 倍（最低 14.5

倍），内存申请-使用比平均 9.6 倍（最低 5.7 倍）——即给环境「名义上」分配了 27.9 份 CPU，实际只用掉 1 份。^[2] 如果能安全地超卖（Overcommit，超卖）这些空闲资源，同样的硬件就能塞下几十倍的环境；反之，成本就是别人的几十倍。

为什么不让环境空闲时直接销毁、用时重建？因为环境是有状态的：改过的文件、装好的依赖、起好的服务都是训练资产，销毁即丢失，重建要把整个任务的历史重演一遍，代价远超空转。为什么不用传统的休眠（Hibernate，休眠）？因为传统虚拟机的内存落盘是秒级到分钟级的，而 RL 的「空闲—唤醒」切换一天要发生成千上万次，休眠的开销会把省下的资源全部吃回去。出路是：让「暂停」和「恢复」便宜到毫秒级，让空闲环境释放 CPU 和可回收内存、有新动作时再原样醒来。官方博客给出的目标是：环境可以在几十毫秒内启动或恢复、几百毫秒内暂停。^[2] 怎么做到的，是本书存储与内存章节的主线。

【零基础知识补给站】

本节前备概念：

- ① **超卖（Overcommit）**：给租户分配的「名义资源」总和超过物理资源总量，赌它们不会同时用满。云计算的经典盈利手段，但前提是负载确实稀疏且错峰。
- ② **申请-使用比（Allocation-to-Usage Ratio）**：官方博客的定义是「分配给所有执行环境的资源总量 ÷ 宿主机上的实际资源用量」，比值越高说明超卖越狠。^[2]
- ③ **推理（Inference）**：用训练好的模型生成输出的过程。Agentic RL 中智能体的每一步「思考」都是一次 GPU 推理，环境必须等它返回。

Insight#1 「大多数环境在空等推理」是整个成本问题的题眼。AgentENV 的全部效率设计——毫秒级暂停恢复、内存回收、资源超卖——都是在把这句观察变现：空闲环境必须便宜。

Insight#2 $27.9 \times / 9.6 \times$ 的超卖比不是优化出来的目标，而是负载的固有属性被测量出来的结果。先测量、再设计，这是系统工程的正确顺序。

2.4 排除法：为什么现有方案都不行

面对「强隔离 + 弹性规模 + 高效率」的三重要求，先别急着看 AgentENV 怎么做，先用排除法（Elimination，排除法）把现成的选项挨个毙掉，设计空间自然就清晰了。

2.4.1 为什么不用容器（Docker）？——隔离太弱

容器与宿主机共享同一个内核，隔离靠命名空间（Namespace）和控制组（Cgroup）这些内核机制「逻辑隔离」。对安分的应用够用，但对奖励驱动、自带 root 权限、有动机主动攻击边界的 RL 智能体远远不够。CSDN

社区分析文章（2026-08-16）的总结很直接：容器隔离不够——agent 带 root，内核共享面太大。^[16] 第二章 2.1

节说过，隔离在这里是训练信号质量问题，不能接受「大概率安全」。

2.4.2 为什么不用传统虚拟机？——太重、太慢

传统虚拟机（如 QEMU/KVM 整机）隔离足够强，但代价是重：启动以秒计，每实例内存开销大，快照是整内存落盘的笨重操作。同一个 CSDN 文章概括为「KVM 整机秒级太慢」。^[16] 官方博客的判断更本质：传统虚拟机难以在隔离性、保真度和资源效率之间取得平衡。^[2] RL 需要毫秒级启停、单机数百实例的密度，传统虚拟机在数量级上就不对。

2.4.3 为什么不用 E2B 这类云沙箱服务？——成本与主权

E2B 等托管沙箱服务开箱即用，但按官方博客的定价对比表（2026-07-26 时点），E2B Sandbox 按「墙钟时间」计费，CPU 约 \$0.0504/vCPU · 小时、内存约 \$0.0162/GB · 小时；Vercel Sandbox 约 \$0.128/vCPU · 小时。^[2]

按申请资源或长期预留计费方案，估算成本是 AgentENV 的 8.8 至 31.7 倍。^[2] 对需要常年维持数千个环境的后训练团队，这是百万级的年度差价。另一层考量是数据主权：训练环境里有未发布的代码仓库与评测题，头部实验室不可能把它放进第三方云。AIRUN 的分析（2026-07-29）也提醒了硬币的另一面：中小企业绝大多数不需要自建这套系统，它解决的是「同时开成千上万个沙箱跑

RL」的规模问题；真正值得学的是「先在沙箱量化验证、再上线」的工程纪律。^[17]

2.4.4 排除之后剩下的设计空间

容器太弱、整机太重、云服务太贵——排除法把设计空间压缩到一个很窄的区间：需要一种**隔离强度接近虚拟机、开销接近容器、可自托管**的执行单元。这个区间里恰好有一个现成答案：Firecracker microVM（微虚拟机），AWS 为 Lambda 打造的轻量虚拟化技术。CSDN 文章的概括即 AgentENV 的配方：Firecracker microVM 隔离 + OverlayBD 懒加载 + 快照/fork 复用，三件事拧在一起。^[16] 下一章我们就来看这个配方的全貌。

【零基础知识补给站】

本节前备概念：

① Namespace / Cgroup：Linux

容器的两块基石，前者隔离「看得到什么」，后者限制「能用多少资源」，但内核始终共享。

② QEMU / KVM：Linux 上经典的虚拟机组合，KVM 提供硬件虚拟化接口，QEMU

模拟整机设备——功能全但重。

③ microVM（微虚拟机）：只保留最小设备模型的轻量虚拟机，保留硬件级隔离边界，砍掉整机的笨重。Firecracker 是代表作。

④ 墙钟时间（Wall-Clock Time）计费：从环境创建到销毁的实际经过时长计费，空闲也照付——对「大部分时间在空转」的 RL 负载最不友好。

Insight#1 排除法的价值在于暴露设计空间的真实形状：三个挑战分别否决了三类现成方案，剩下的交集（硬件级隔离 + 毫秒级启停 + 按需供给 + 自托管）就是 AgentENV 的坐标。

Insight#2 判断一个方案行不行，别只看「能不能跑」，要看它在 RL 的特定节奏下「贵不贵、纯不纯」：节奏错配（分钟级拉镜像、墙钟计费）和信号污染（弱隔离）都是数量级层面的否决项，不是优化能弥补的。

第三章 AgentENV 全景：定位、数字与技术选型

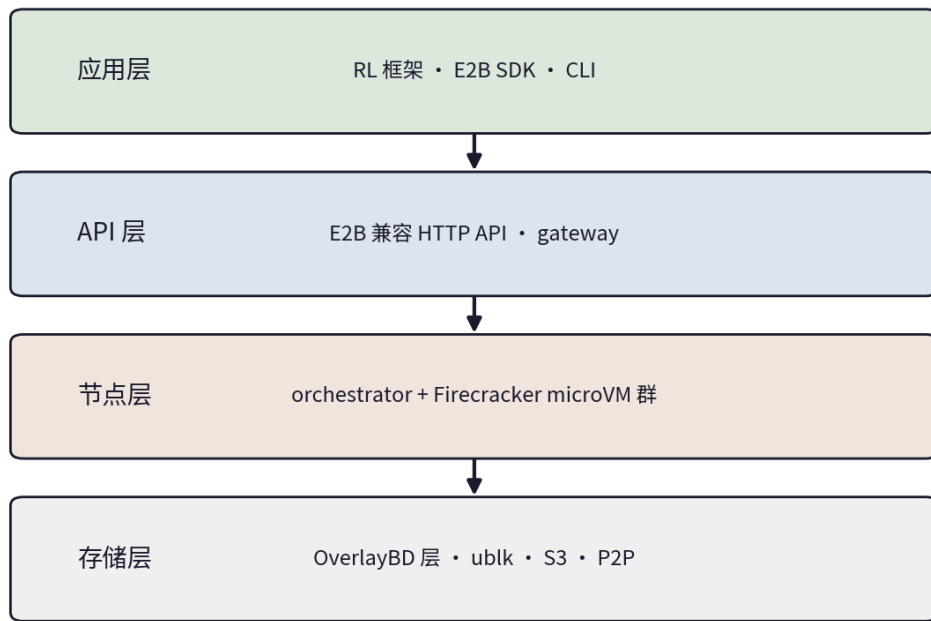
前两章解决了「为什么需要新系统」，本章给出新系统的第一印象：它是什么（定位）、做到了什么（关键数字）、靠什么做到（技术选型）、以及代码在哪里（开源地图）。本章是全书的「总目录」——这里点到为止的每一项技术，后续章节都会逐一拆到底。

3.1 一句话定位与四大设计支柱

官方博客的定位只有一句话：AgentENV (AENV) 是一个面向大规模 Agentic RL 的智能体执行环境 (Agent Execution Environment)。^[2] README 的表述互为补充：一个大规模运行智能体环境的平台 (Running agent environments at scale)，为 Kimi K3 的 Agentic RL 训练提供支撑。^[1] 注意两个关键词：一是「执行环境」而非「沙箱」——它管理的不是一次性隔离容器，而是有状态、可快照、可分叉的训练资产；二是「大规模」——设计的及格线不是「能跑」，而是数千到数万个并发环境下的成本与稳定性。

官方博客的 Core Design 一节给出了系统的四大设计支柱，它们恰好逐一回应第二章的三大挑战（支柱三同时服务效率与 RL 算法需求）：^[2]

- **支柱一：跨节点扩展异构环境 (Scaling Heterogeneous Environments Across Nodes)**。在多服务器上大规模运行 Firecracker microVM，兼容 OCI 镜像生态；镜像通过 OverlayBD 从远端对象存储或远端文件系统按需加载，本地磁盘只作有限容量的热数据缓存。于是集群可用的镜像与环境模板总量可以超过单节点磁盘容量，也无需在每台宿主机预热所有镜像。——回应挑战二「弹性规模」。
- **支柱二：让空闲环境变便宜 (Making Idle Environments Inexpensive)**。基于快照的环境可在几十毫秒内启动或恢复、几百毫秒内暂停；环境等待模型推理时释放 CPU 与可回收内存，新动作到达后从原状态快速恢复。训练系统得以用更低成本维持数千个逻辑上活跃的有状态环境。——回应挑战三「高效率」。
- **支柱三：原生支持增量快照与分叉 (Native Support for Incremental Snapshots and Forks)**。增量记录内存与文件系统的变化，保存状态时不必复制整台虚拟机；快照可持久化到 S3 兼容对象存储或共享分布式文件系统，节点故障不丢状态；同一中间状态还能以写时复制 (Copy-on-Write, CoW, 写时复制) 分叉成多个独立环境，让智能体并行尝试不同的工具调用、代码修改或操作序列，为多轨迹采样与树搜索 (Tree Search, 树搜索) 提供高效的环境底座。——这是 RL 算法层的「杀手锏」特性。
- **支柱四：长期保持高密部署 (Maintaining High Density Over Time)**。microVM 持续运行时，每个环境会逐渐积累私有状态，部署密度随时间下滑；AgentENV 持续复用与回收存储和内存来对抗这种下滑：存储侧用只读 OverlayBD 层 + 内容寻址缓存去重，每个 microVM 只存自己的增量写入，磁盘 I/O 走 ublk、io_uring 与 Direct I/O 减少虚拟机后端与宿主机之间的冗余缓存；内存侧让同源模板恢复的环境共享只读内存快照，干净页经由宿主机 Page Cache 复用，并用 DAMON Reclaim 与 Firecracker Balloon 持续回收客户机内的文件缓存。^[2]



自顶向下：用户接口 → 兼容网关 → 单机虚拟化执行 → 镜像与块存储底座

图 3-1 AgentENV 全景：四大设计支柱立于 Firecracker microVM 与高性能 I/O 底座之上，兼容 OCI 镜像与 E2B API 生态（据官方博客 Core Design 一节整理^[2]）

为什么不把这四件事做成四个独立工具？因为它们是同一个状态机的四个侧面：镜像按需加载决定了「从哪启动」，快照决定了「如何保存」，分叉决定了「如何复制」，暂停/回收决定了「如何省钱」——拆开后每一环都拿不到端到端的毫秒级体验。mushroom.cv

的社区分析（2026-07-28）对此有一个精辟的范式判断：AgentENV 把 RL 训练环境从「服务」模型转成了「状态机 +

快照」模型——环境可以快速序列化/反序列化的状态机，快照是 checkpoint，分叉是从 checkpoint 并行探索，暂停是不销毁状态地出让资源。^[19]

【零基础知识补给站】

本节前备概念：

① **写时复制（CoW）**：复制数据时先不真正拷贝，父子共享同一份，谁要修改才单独拷贝被改的部分。让「复制一台电脑」从分钟级降到毫秒级的关键思想。

② **增量快照（Incremental**

Snapshot）：只记录「自上次以来变化的部分」的快照，与全量快照相对。

③ **树搜索（Tree**

Search）：从同一状态出发并行探索多条动作路径再择优的算法，需要廉价的环境分叉能力。

④ **Page Cache（页缓存）**：操作系统把读过的文件数据缓存在内存里的机制，多台虚拟机读同一镜像时可以共享。

Insight#1 四大支柱的逻辑链值得背诵：镜像懒加载解决「装得下」，毫秒暂停恢复解决「养得起」，增量快照与分叉解决「RL 算法用得上」，持续回收解决「跑得久」。缺任何一环，另外三环的价值都会打折。

Insight#2 「状态机 + 快照」范式是比任何单项技术更重要的收获：一旦环境被视为可序列化的状态机，checkpoint、分叉、暂停这些操作就变成了状态机上的自然运算，而不是外挂功能。

3.2 关键数字：先记住这张表

AgentENV

官方披露的数字不多，但每一个都打在第二章的痛点上。下表逐字整理自官方博客与 README，建议读者先建立量级直觉，后续章节会逐个解释这些数字「为什么能做到」。

指标	数值	含义与出处
模板启动/恢复延迟	低至 49 ms	从模板或快照把一台「电脑」拉到可用 ^[2]
快照创建延迟	低至 133 ms	增量记录内存与文件系统变化 ^[2]
并发分叉摊销延迟	低至 122 ms	扇出场景下每个立即可执行的子环境 ^[2]
单机并发环境	实测 400 个	128 核 CPU / 512 GB 内存，环境规格 2 vCPU/4 GB 或 4 vCPU/8 GB ^[2]
集群规模	3 万个环境	生产集群已验证，可随节点线性扩展 ^[2]
成本降幅	88.6%–96.8%	典型负载下对比现有方案 ^[2]
CPU 申请-使用比	平均 27.9×，最低 14.5×	70 节点、约 22.5 万环境的生产数据 ^[2]
内存申请-使用比	平均 9.6×，最低 5.7×	同上 ^[2]
生产镜像规模	150 万个	K3 技术报告 ^[3]
月度成本对照	约 ¥15,300 vs 8.8–31.7 倍	300 环境（4 vCPU/8 GB）连续 720 小时 ^[2]

表 3-1 AgentENV 关键数字一览（数据逐字取自官方博客与 K3 技术报告）

三组数字值得单独停留。第一组是**延迟三兄弟**：49 ms 启动、133 ms 快照、122 ms 摊销分叉，它们共同把第二章说的「创建环境、执行任务、保存状态、恢复或复制环境」这条耗时昂贵的循环压缩到了毫秒级。README 的口径更保守（快照式启动/恢复 50 ms 内、暂停 100 ms 内、重磁盘写场景快照 100 ms 内），与博客互为印证。^[1]

官方博客还强调：与现有方案相比，AgentENV 把环境生命周期管理开销降低了大约一个数量级。^[2]

第二组是**规模**：单机 400

个环境意味着一台标准高配服务器就能装下一个中小团队的全部训练环境；集群 3 万个且可线性扩展，则回答了「能不能撑住头部实验室的后训练」。^[2] openEuler 社区的分析文章（2026-08-11）评价：对测试场景这是降维打击——「黄金镜像」只需构建一次，之后每

次从快照恢复，环境准备从秒级降到毫秒级。^[18]

第三组是**成本**：300 个 4 vCPU/8 GB 环境连续跑 720

小时，按实测最低资源申请-使用比估算，AgentENV 月度资源成本约 ¥15,300；同样的 CPU/内存申请假设下，按请求计费或长期预留资源的方案估算成本是 AgentENV 的 8.8 至 31.7 倍（对比不含容器实例）。^[2]这个数字不是营销话术，而是 2.3 节「资源稀疏性」的直接变现。

读这张表时还要记住两条方法论。其一，「低至 49 ms」这类措辞是**最佳情形**：模板与快照已在本地缓存、脏数据最少时的下限；真实负载下的完整延迟会更高，社区第三方实测（见第 6 章）会给出脏内存量与 fork 延迟的定量关系。其二，所有数字都标注了测量条件（128 核/512 GB 机型、70 节点高负载时段、720 小时连续运行），脱离条件引用数字是本报告坚决反对的做法——读者在别的场合转述时也应保留这些前提。

【零基础知识补给站】

本节前备概念：

- ① **摊销（Amortized）延迟**：一次扇出创建多个子环境时，总开销平摊到每个子环境上的延迟，比单个创建更能反映并发场景的真实成本。
- ② **vCPU**：虚拟 CPU，云平台切分物理 CPU 的计量单位。
- ③ **线性扩展（Linear Scalability）**：加一台机器就多一份容量，性能不随集群变大而衰减——无共享瓶颈架构的标志。

Insight#1 读系统论文/博客时，数字要成组记忆：AgentENV 的「49/133/122 ms」对应启停存取四个动作，「400/30000」对应单机与集群两个尺度，「27.9×/9.6×」对应 CPU 与内存两种资源——三组数字就是整个系统的性能画像。

Insight#2 所有数字的共同指向是同一句话（官方博客原文）：让研究者和开发者把算力花在模型训练本身，而不是花在环境管理上。^[2]

3.3 技术选型总览：每件兵器为什么是它

下表把 AgentENV

的技术栈一次摊开。此刻只需记住每项选型「解决了哪个挑战」；机制细节（ioctl 怎么发、层怎么叠、页怎么回收）留给后续对应章节。

技术	角色	为什么选它（一句话）	详解
Firecracker	隔离执行单元	microVM 提供硬件级隔离边界、毫秒级启动，卡在「容器太弱、整机太重」的甜蜜点上（2.4 节排除法的唯一幸存者）	第 4 章
OverlayBD	镜像按需加载	把 OCI 镜像变成可按块远程懒加载的块设备，镜像集合因此可以超过单盘容量数个数量级 ^[1]	第 5 章
ublk	磁盘设备路径	用户态块设备框架，让自研的懒加载/分层逻辑不用写内核模块就能以内核块设备的身份被虚拟机使用 ^[2]	第 5 章

技术	角色	为什么选它（一句话）	详解
io_uring + Direct I/O	I/O 加速	异步无锁 I/O 加绕开冗余缓存，减少虚拟机后端与宿主机之间的重复缓存开销 ^[2]	第 5 章
DAMON Reclaim + Balloon	内存回收	持续识别并回收客户机内冷内存与文件缓存，支撑 9.6× 的内存超卖 ^{[1][2]}	第 7 章
增量快照 + CoW 分叉	状态管理	只记变化、分叉共享，把「保存/复制一台电脑」压到百毫秒级，服务多轨迹采样与树搜索 ^[2]	第 6 章
Rust	核心语言	性能贴近 C 且内存安全，敢在数据路径上长期跑；仓库实测 443 个 Rust 源文件 ^[1]	第 4-7 章
Go	集群服务语言	调度器、网关、API 等控制面服务需要快速开发与成熟生态；仓库实测 43 个 Go 源文件 ^[1]	第 8 章
E2B 兼容 API	生态接口	指向 E2B_API_URL 即可复用标准 E2B SDK、代码零改动，迁移成本近乎为零 ^{[1][19]}	第 9 章

表 3-2 AgentENV 技术选型总览（「详解」列为本报告后续章节编号）

为什么不一上来就用最「标准」的组件拼？比如用 containerd 原生快照器存镜像、用 QEMU 跑虚拟机、用 Kubernetes 原生调度一切。因为第二章的排除法已经说明：标准组件是为无状态服务负载优化的，在「有状态 + 毫秒级 + 十万级镜像」的约束下，每一层都需要重新设计。但这并不意味着闭门造车——AgentENV 对生态的兼容非常克制：镜像格式兼容 OCI、快照可落 S3 兼容对象存储、API 兼容 E2B，都是「内核自研、接口从众」的策略。^{[1][2]}

【零基础知识补给站】

本节前备概念：

① 用户态 /

内核态：程序运行在受限制的「用户态」，操作系统核心运行在特权「内核态」；ublk 的巧妙在于让用户态程序提供块设备服务，兼顾灵活与安全。

② **块设备（Block Device）：**以固定大小数据块读写的存储设备抽象，硬盘、云盘都是。虚拟机看到的「磁盘」就是一个块设备。

③ 控制面 /

数据面：控制面负责调度决策（这台环境放哪个节点），数据面负责实际读写执行。AgentENV 用 Go 写控制面、Rust 写数据面，是常见的性能分工。

Insight#1 技术选型的统一审美是「内核自研、接口从众」：数据路径上每一层都为 RL 负载重写，对外接口（OCI/S3/E2B）却尽量兼容标准——前者决定性能上限，后者决定生态成本。

Insight#2 记选型不如记「为什么是它」：Firecracker

是排除法的幸存者，ublk/OverlayBD/io_uring

是「懒加载块设备」这条主线的三件套，DAMON/Balloon 是超卖承诺的兑现机制。

3.4 开源情况与仓库地图

AgentENV 以 MIT 协议开源（商用无授权障碍^[17]），仓库位于 GitHub 的 kvcache-ai 组织下。社区热度可以作为参考：AIRUN 统计截至 2026-07-29 约 1475 star、128 fork，并提醒仓库还很年轻；^[17] CSDN 文章记录开源三周即达 3.2K star（2026-08-16）。^[16] 本报告撰写时对仓库实测：Rust 源文件 443 个、Go 源文件 43 个，与「约 90% Rust」的社区观察一致。

仓库的顶层目录即系统的架构轮廓，导读如下（第 8、9 章会逐层深入）：

目录	内容	阅读建议
src/	Rust 核心：api、orchestrator、sandbox、snapshot、template、image、overlaybd、p2p、volume、setup、observability 等模块	数据面主战场，第 4-7 章的主读区
crates/	Rust 周边 crate：aenv（CLI）、warm-pool（预热池）、linux-cap、benchmarks、e2e-tests、object-store-operator、shell-util 等	按需查阅，warm-pool 与 benchmarks 尤其值得看
services/	Go 控制面：api、gateway、scheduler、config、shared	集群调度与服务化，第 8 章主读区
deploy/ · config/	Docker Compose / Kubernetes 部署与配置样例	动手部署时对照 README 使用
docs/ · scripts/	官方文档源与安装脚本（install.sh 等）	快速上手入口
storage/ · containerd-plain-snapshotter/ · thirdparty/	存储相关组件与第三方依赖	第 5 章涉及
tests/ · tools-image/	测试与工具镜像	理解工程质量与测试策略

表 3-3 AgentENV 仓库地图（据 kvcache-ai/AgentENV 仓库实测整理^[1]）

上手路径在 README 里写得很平实：install.sh 装服务端与 aenv CLI、systemctl start aenv 起服务、aenv auth 粘贴首次启动生成的 API key、aenv pull 拉模板，然后 aenv start / exec / pause / resume / timeout / delete 完成环境全生命周期操作。^[1] 前置条件是 Linux 内核 6.8+ 与 /dev/kvm 访问权（不支持标准 KVM 的机器有 PVM 部署指南）。^{[1][17]}

【注意】

安全提示（README 原文警告）：AgentENV 对 API 请求做认证，但**不加密流量**。不要在不可信的明文网络上发送 API key；请在可信网络中运行，或在反向代理/负载均衡处终结 HTTPS。^[1] AIRUN 的分析也特别强调了这一点：不可暴露公网。^[17]

为什么不把代码结构留到读者自己探索？因为 443 + 43 个文件的仓库对零基础读者并不友好，而「先知道哪层住在哪里」能把后续章节的阅读成本降低一个量级。至此，背景（第一章）、动机（第二章）与全景（第三章）已经就绪——从下一章开始，我们钻进 Firecracker microVM 的内部，看毫秒级启动的第一块基石如何砌成。

【零基础知识补给站】

本节前备概念：

- ① **MIT 协议**：最宽松的开源协议之一，允许自由使用、修改、商用，只需保留版权声明。
- ② **crate**：Rust 的代码包单位，一个仓库可含多个 crate（workspace 结构）。
- ③ **/dev/kvm**：Linux 内核暴露的硬件虚拟化设备接口，Firecracker 通过它创建 microVM。
- ④ **CLI / API / SDK**：命令行工具、编程接口、语言封装库——AgentENV 三层都提供，且 API 兼容 E2B。

Insight#1 开源让 AgentENV 的价值翻倍：88.6%–96.8% 的成本降幅和 3 万环境规模不再是「信不信由你」的宣传数字，而是每一行都可审阅、可复现、可魔改的工程事实。

Insight#2 「Rust 写数据面、Go 写控制面」的双语结构本身就是一条架构信息：读源码时先分清自己在哪一面，数据面的每一行都关乎毫秒级延迟，控制面的每一行都关乎万级规模的正确性。

第四章 使用方式：从安装到第一个沙箱

本章以「零基础上手」为视角，走完从安装 AgentENV（下称 AENV）到启动第一个沙箱的完整链路：前置条件检查 → 安装 → 认证 → CLI 命令树 → HTTP API 与 E2B 兼容接入 → 配置体系 → 沙箱内通信机制。所有结论均来自仓库源码^[1]与官方文档，命令序列与社区实测文章 [16] 一致。

4.1 前置条件与安装

AENV 对宿主环境的要求非常明确：Linux 内核 6.8+，并且能访问 /dev/kvm（KVM，Kernel-based Virtual Machine，内核级虚拟机接口），因为沙箱全部跑在 Firecracker microVM 里。如果服务器不支持标准 KVM（例如云厂商的嵌套虚拟化受限），可以走 PVM（PVM: Efficient Shadow Paging 论文提出的 KVM 兼容虚拟化方案，需安装 pvm-kernel-6.12.33 内核并加载 kvm_pvm 模块），但 PVM 仍是 EXPERIMENTAL，本章一笔带过，细节见部署章节。

安装有两条路径。路径 A 是一行脚本安装服务端加 aenv CLI，并注册 systemd 服务：

```
curl -fsSL https://raw.githubusercontent.com/kvcache-ai/AgentENV/main/scripts/install.sh |
sudo bash
sudo systemctl start aenv
```

路径 B 是 Docker 方式，适合不想改动宿主环境的场景：

```
curl -fsSL https://raw.githubusercontent.com/kvcache-ai/AgentENV/main/scripts/docker-setup.sh |
sudo bash
docker pull ghcr.io/kvcache-ai/aenv-server:latest
docker run -d --name aenv-server --privileged -v /dev:/dev -p 8000:8000 \
  ghcr.io/kvcache-ai/aenv-server:latest
```

服务端默认监听 `http://127.0.0.1:8000`。CLI 也可以独立安装（比如 CLI 装在你的笔记本、服务端在远端机房），支持 Linux 和 macOS 的 x86_64 与 arm64：

```
curl -fsSL https://raw.githubusercontent.com/kvcache-ai/AgentENV/main/scripts/install-cli.sh |
bash
```

注意 Docker 方式里容器以 `--privileged` 运行并挂载整个 `/dev`——这是 `ublk`（userspace block device，用户态块设备）与 `netns`（network namespace，网络命名空间）操作的必然要求，不是偷懒。为什么不提供普通容器镜像？因为 AENV 服务端需要创建网络命名空间、加载 `ublk` 设备、访问 `/dev/kvm`，这三件事都无法在非特权容器里完成，这是架构决定的硬约束。

【零基础知识补给站】

KVM 是 Linux 内核自带的虚拟化模块，Firecracker 是建立在 KVM 上的轻量级虚拟机监视器（VMM，Virtual Machine Monitor），每个 AENV 沙箱就是一个 Firecracker microVM。`/dev/kvm` 是用户态程序访问 KVM 能力的设备文件。可以用 `ls -l /dev/kvm` 确认它存在且可访问；用 `uname -r` 确认内核版本不低于 6.8。

Insight#1 安装脚本与 systemd 的组合意味着 AENV

定位是「常驻服务」而非「按需工具」——它预热的 Firecracker 进程池、`ublk daemon` 都是长驻进程，一次安装长期受益。

Insight#2 社区实测文章 [16] 给出的上手序列与官方 README 完全一致：`install.sh` → `systemctl start aenv` → `aenv auth` → `aenv pull` → `aenv start/exec/pause/resume/timeout/delete`，说明命令面非常稳定。

4.2 认证：API key 的三级来源

AENV 服务端首次启动时会自动生成一个 API key（API 密钥），所有控制面请求都要在 `X-API-Key` 头里携带它。密钥解析逻辑在 `src/api_key.rs` 的 `ApiKey::resolve_from`（L56–87），按优先级依次尝试三个来源：

优先级	来源	说明
1（最高）	环境变量 <code>AENV_API_KEY</code>	显式指定，适合临时覆盖与 CI
2	外部 secret 文件 <code>/run/secrets/api-key</code>	K8s 风格挂载，允许符号链接，但必须是 regular file
3	托管密钥 <code>~/.aenv/secrets/api-key</code>	不存在时用系统 RNG 生成 32 字节随机数，hex 编码后加 <code>e2b_</code> 前缀

表 4-1 API key 的三级来源（`src/api_key.rs` L56–87）

托管密钥的生成是并发安全的：写盘用 `persist_noclobber`（`managed_secret.rs` L106–120），两个进程同时首次启动时，后者只会读到前者生成的 key（`CreateOutcome::Existing` 分支），不会各写一份互相覆盖。文件权限被强制收紧到 0600（仅属主可读写），父目录必须是名为 `secrets` 的 0700 目录、非符号链接、属主为当前

eid；打开时使用 O_NOFOLLOW|O_NONBLOCK 防符号链接与 FIFO 攻击（managed_secret.rs L127–220）。这是全仓库安全工程密度最高的文件。

拿到 key 后（原生安装用 `sudo cat /var/lib/aenv/secrets/api-key`，Docker 用 `docker exec aenv-server cat /workspace/env/secrets/api-key`），执行交互式认证：

```
aenv auth
# AENV server URL [http://localhost:8000]: http://127.0.0.1:8000
# API key: <粘贴上面读到的 key>
```

aenv auth 用 rpassword 隐式输入（不回显），把凭据存成 TOML 到系统标准目录的 aenv/credentials 文件并 chmod 600。为什么不把 key 直接塞在配置文件里用明文？——它确实是明文，但 0600 权限加 ProjectDirs 标准位置已经是 CLI 工具的业界惯例（与 docker login 的 ~/.docker/config.json 同思路）。服务端比对时使用 subtle::ConstantTimeEq 常数时间比较（api_key.rs L52–54），Debug 输出统一替换为 [REDACTED]，防止日志泄漏。

【注意】

README 明确警告：AENV 对 API 请求做认证但不加密流量，不要在不可信的明文网络上传输 API key，应部署在可信网络或在反向代理/负载均衡上终结 HTTPS。社区文章 [18] 也曾指出早期版本不支持认证，新版本已补上认证但加密仍要用户自己负责。

【零基础知识补给站】

「常数时间比较」（constant-time comparison）指比较两个字符串耗时不随第一个不同字节的位置变化，防止攻击者通过计时侧信道逐字节猜出密钥。普通 == 短路比较会在第一个不同字符处提前返回，泄漏信息。

Insight#1 三级密钥来源的设计明显向 K8s 部署场景倾斜：/run/secrets/api-key

允许符号链接正是为了兼容 Kubernetes Secret 挂载的 ..data

软链机制，且仓库里有专门的符号链接测试（api_key.rs L194–210）。

Insight#2 「不存在就生成」的兜底策略把「第一次部署」的门槛降到零——不需要管理员先配置密钥，服务自己会长出一个安全的随机密钥。

4.3 CLI 全命令树逐条讲解

aenv CLI 源码在 crates/aenv/，命令树覆盖认证、模板、沙箱全生命周期、构建与体积最大的 codex 子模块（约 2800 行，用于在沙箱中运行 Codex agent）。完整命令清单如下：

命令	作用	示例
auth	交互式配置服务端 URL 与 API key	aenv auth
pull	从 OCI 镜像构建模板（FROM image → template）	aenv pull ubuntu:22.04 --name ubuntu
build	本地 Dockerfile 经 wss:// BuildKit 隧道构建模板	aenv build . --name myenv
template	模板管理（list/ls 等）	aenv template ls

命令	作用	示例
start	从模板启动沙箱并附着交互式 shell	aenv start ubuntu / aenv start ubuntu --detach
connect (cn)	重新附着已存在沙箱的 shell	aenv cn
exec	在沙箱内执行一次性命令	aenv exec ls -la /
list (ls)	列出沙箱，TTY 输出表格，管道时输出 JSON	aenv ls --output json
pause	暂停沙箱（快照+停 VM）	aenv pause
resume	从暂停恢复	aenv resume
timeout	调整 TTL（存活时间）	aenv timeout 600
snapshot (snap)	捕获并发布快照	aenv snapshot
volume	持久卷管理	aenv volume create ...
codex	在沙箱中跑 Codex agent	aenv codex ...
delete (rm)	删除沙箱	aenv rm
upload / download / completion	文件传输与 shell 补全	aenv upload local.txt /remote/path

表 4-2 aenv CLI 命令树（crates/aenv/）

上手四连，来自 README 快速开始^[1]：

```
aenv pull ubuntu:22.04 --name ubuntu # 拉取并构建模板
aenv start ubuntu          # 启动沙箱并附着 shell
aenv exec <sandbox-id> ls -la /    # 一次性命令
aenv timeout <sandbox-id> 600     # TTL 延长到 600 秒
```

两个实现细节值得知道：其一，CLI 内部是双 HTTP 客户端——同步用 ureq、异步用 reqwest，统一注入 X-API-Key 头，连接超时 5 秒、请求超时 120 秒，收到 401 会提示 "run `aenv auth`"。其二，main.rs 手动安装 rustls 的 aws-lc-rs provider，因为 ureq 用 aws-lc-rs 而 reqwest 用 ring，两套加密后端并存需要显式消除歧义——这类「依赖缝合」痕迹是一个真实演进中的项目的常态。

【零基础知识补给站】

TTL（Time To Live，存活时间）是沙箱的自动过期机制：到期后按 timeout_action 自动 Pause 或 Delete。默认沙箱寿命只有 15 秒（default.toml 中 [orchestrator] 配置），这是为 RL 训练「用完即弃」场景设计的，交互使用记得用 aenv timeout 续期。

Insight#1 「TTY 输出表格、管道输出 JSON」的约定（`aenv list --output table|json`）让 CLI 既能给人看也能被脚本消费，是 Unix 工具的良好传统。

Insight#2 `codex` 是命令树中最大的子模块（约 2800 行），暗示「在沙箱里跑编码 agent」是一等公民场景而非事后补丁。

4.4 HTTP API 与 E2B 兼容

如果不用 CLI，AENV 的全部能力也暴露在 HTTP API 上。路由表由 OpenAPI 生成代码产生（`src/api/generated/src/server/mod.rs` L46–158，crate 名 `agentenv_http_server`），整体是 E2B API 的兼容实现（v1 路径、响应模型、连 API key 前缀 `e2b_` 都是 E2B 形状），外加 AENV 自有的 v2/v3 扩展。E2B（`e2b.dev` 的云端沙箱服务）兼容意味着已有 E2B SDK 用户零改代码迁移——只需设置三个环境变量：

```
export E2B_API_URL=http://127.0.0.1:8000 # 指向你的 AENV 服务
export E2B_SANDBOX_URL=... # 沙箱流量入口
export E2B_API_KEY=<你的 key> # AENV 生成的 key 前缀就是 e2b_
# 之后 E2B Python/TypeScript SDK 的 Sandbox.create/list/commands.run/pause/kill 直接可用
```

兼容的实现分三层叠加，而不是单独写一层适配器：

- API 形状兼容：OpenAPI 定义本身就是 E2B schema，连 `client_id` 这种「仅为 E2B Python SDK 保留的废弃字段」都保留（`impls/sandbox.rs` L344/364/428 注释明言）；鉴权头是 E2B 的 `X-API-Key`（`auth.rs` L13）。
- 鉴权适配：生成 router 自带的 `claim` 抽取被实现为「外层中间件是权威」的透传适配器（`auth.rs` L137–163），避免生成层拒绝 AENV 的 API-key 请求。
- 流量代理兼容：沙箱内 `envd` 与用户端口通过 `/proxy/*` 前缀路由或 Host 头域名路由转发到沙箱 `netns` IP，代理层同时认 `x-agentenv-sandbox-id` 和 `e2b-sandbox-id` 两套头（`proxy.rs` L86–92 的 `SANDBOX_ID_HEADER/E2B_SANDBOX_ID_HEADER` 双套常量），旧 SDK 与新客户端都通吃。

常用 v1 端点一览：POST `/sandboxes` 热创建、POST `/sandboxes-cold` 冷创建、POST `/sandboxes/{id}/pause|resume|fork|snapshots|timeout`、DELETE `/sandboxes/{id}` 等，v2 增加分页游标与日志查询，v3 是模板异步构建。为什么不做成纯自研 API？因为 E2B 兼容是生态杠杆：社区文章 [20] 评价「E2B 兼容 API 使迁移成本近零」，AENV 用兼容换取了现成的 SDK、文档与心智模型。

【零基础知识补给站】

OpenAPI 生成（codegen）指从一份 OpenAPI 规格文件自动生成服务端路由骨架与模型类型，handler 只做参数校验后转调真实实现。AENV 规定生成代码只能改 schema 再加 make 重新生成，不允许手改生成物（`CONTRIBUTING.md`），避免「改丢失」。

Insight#1 双套头设计 (x-agentenv-*/e2b-*) 是兼容的「最低成本方案」：不fork协议、不加适配层，只多认一组头，维护成本近乎为零。

Insight#2 代理层还有隐藏功能：auto-resume——请求命中已暂停沙箱时自动 resume 再转发 (proxy.rs try_auto_resume L785, 超时 60s)，对调用方完全透明。

4.5 配置体系导读：default.toml 最重要的 12 项

AENV 的配置集中在 config/default.toml，进程级单例

ConfigManager (src/cfg.rs, OnceLock 包裹) 在启动时读入，之后所有子系统经

ConfigManager::global_config() 访问。配置项很多，按类别挑出最重要的 12 项：

类别	配置项	默认值/含义
全局路径	home_path / runtime_path	/var/lib/aenv 与 /run/aenv，支持 \$AENV_HOME 占位符
虚拟化	virtualization_mode	"kvm" 或 "pvm"；快照只能在创建模式下恢复
firecracker	socket_timeout_secs	3；API socket 就绪轮询超时
image.resolver	default_image	ubuntu:24.04；未指定镜像时的默认 rootfs
image.cache	capacity_gb / GC 水位	100 GB 有界缓存；0.95 触发 GC、清到 0.70
network.egress	always_denied_cidrs	默认拒绝 RFC1918 私网/链路本地地址的出站
machine	memory_mb / vcpu	1024 MiB / 2 vCPU 的默认沙箱规格
orchestrator	默认沙箱寿命 / 驱逐周期	15 s TTL；自动驱逐每 1 s 扫描一次
snapshot	repository_backend	posix_fs 或 oss；publish_compression 支持 lz4/zstd
ublk	daemon metrics 端口	:9103 独立暴露 ublk 指标
pool	预热池低/高水位	2 / 64；network、block、firecracker 三池
template_build	单节点并发 / BuildKit 镜像	4 (超出返回 429) ；moby/buildkit:v0.33.0

表 4-3 default.toml 最重要 12 项速查 (config/default.toml)

读配置表的要点是看出设计取向：默认 TTL 15 秒、GC 水位 0.95→0.70、预热池 2/64，这些数字都在说同一件事——为「高吞吐、短生命周期、环境即耗材」的 RL 训练负载调优，而非为长驻开发环境。为什么不提供运行时 reload？OnceLock 单例意味着配置一次读入终身不变，换来的是任意子系统随时 global_config() 零成本访问的简洁，代价是改配置必须重启——控制面重启很快，这个取舍成立。

【零基础知识补给站】

OOM 式 GC 水位：image.cache 的「capacity 100 GB、用到 95% 触发清理、清到 70% 停手」是经典的滞回 (hysteresis) 设计——两个阈值拉开距离，避免在单阈值附近反复抖动清理。

Insight#1 `allowed_registries`（镜像仓库白名单）是安全配置，防止有人让服务拉取任意仓库的恶意镜像。

Insight#2 `orchestrator` 里 `auto-resume` 最小 300 s 的下限防止「秒级频繁唤醒」把 `pause/resume` 变成 CPU 风暴来源。

4.6 沙箱内通信：envd 与 ConnectRPC

沙箱启动后，CLI/SDK 如何往里面发命令？答案是每个沙箱内有一个常驻 agent 叫 `envd`，负责进程执行、文件读写、PTY 终端等控制面操作。对外通信协议不是原生 `gRPC`，而是手写实现的 `ConnectRPC`（`Connect-RPC` 是 Buf 公司提出的协议，`unary` 调用就是普通 HTTP POST，`server-streaming` 用 `chunked envelope` 编码）[23]。

为什么不用标准 `gRPC`？用「为什么不 X」排除法：标准 `gRPC` 依赖 HTTP/2 的帧与流语义，而 AENV 的反向代理链路（CLI → 节点 `axum` / `proxy` → 沙箱 `netns`）不做 HTTP/2 到 `envd` 的终结与转发。要在 HTTP/1.1 友好的代理上跑流式 RPC，只剩三条路：(a) 给代理加 HTTP/2 支持——工作量巨大且引入新复杂面；(b) 退回 `WebSocket` 自定义协议——丢掉 RPC 生态；(c) 用 `ConnectRPC`——天然 HTTP/1.1 兼容、可穿代理，协议开销又接近 `gRPC`。AENV 选了 (c)，客户端在 `crates/aenv/src/grpc/mod.rs` 手写 `Connect` 客户端，路由靠 `x-agentenv-sandbox-id` 与 `x-agentenv-target-port`（`envd` 端口 49983）两个头。

这套选择的效果：任何 HTTP/1.1 反向代理（包括 `nginx` 和 AENV 自家 `gateway`）都能透明转发沙箱控制流量，`server-streaming` 照样可用（PTY 交互式 shell 依赖它）。服务端也有配套优化：`axum` 监听 `socket` 对每个连接 `set_nodelay(true)`（`server.rs` L183–187），注释解释 `Connect-RPC` 流是一连串小帧，`Nagle` 算法会给每个短命令加约 40ms 的地板延迟——这在 RL 高频短命令场景是不可接受的。

【零基础知识补给站】

`Nagle` 算法会把小 TCP 包攒起来一起发以降低包头开销，代价是增加延迟；`TCP_NODELAY`（`set_nodelay`）关闭该算法，小包立即发出。交互式/流式协议通常都要关 `Nagle`。

Insight#1 「手写 `Connect` 客户端」而非用官方生成器，是因为路由头注入与代理语义是 AENV 特有的，手写反而更薄。

Insight#2 `envd` 就绪是沙箱对外可见 `Running` 的必要条件（`wait_for_ready` 轮询 `envd` 并 `init env/workdir/user`），通信协议的选择直接处在启动热路径上，HTTP/1.1 兼容性因此是硬需求而非偏好。



一次典型会话的生命周期：认证 → 拉镜像 → 起沙箱 → 交互 → 挂起/分支 → 回收

图 4-1 从安装到第一个沙箱的完整流程（示意）

第五章 总体架构：控制面与数据面

本章从进程视角拆解 AENV
单机服务端的总体架构：三层视图、启动序列、沙箱生命周期状态机、create 热路径调用链，以及支撑这一切的并发与可靠性模型。所有行号均基于仓库工作副本（agentenv-repo/）^[1]。

5.1 三层视图与进程边界

AENV 的系统可以画成三层：最上是集群控制面（cluster control plane）——Go 写的 Gateway 与 Scheduler 服务，负责多节点路由与调度（第十一章详述）；中间是节点服务——一个 Rust 进程 aenv-server，承担单机上的 API、编排、沙箱管理全部职责，是本章主角；最下是存储子系统——overlaybd 块镜像、ublk daemon、镜像缓存与快照仓库（posixfs/OSS/ACR/P2P 后端）。

进程边界很清晰：唯一的服务端入口是 src/bin/server.rs 的 main 函数（L56–242），一个进程内用 tokio 异步运行时承载 axum HTTP 服务、Orchestrator 编排器、后台周期任务。沙箱不是线程而是真正的进程——每个沙箱对应一个独立的 Firecracker 进程，住在自己的网络命名空间里。为什么不把每个沙箱做成独立守护进程管理器？因为单进程编排可以用内存锁与 watch channel 做微秒级状态协调，跨进程就只能引入 etcd/DB 锁这类外部协调器——AENV 刻意避免了后者（见 5.5）。

层	语言/形态	职责
集群控制面	Go 服务（gateway、scheduler）	多节点路由、调度、节点发现
节点服务	Rust 单进程 aenv-server	HTTP API、编排、Firecracker/ublk/网络管理
存储子系统	overlaybd + ublk + 快照仓库	镜像懒加载、块设备、快照持久化与分发

表 5-1 三层架构与职责划分

【零基础知识补给站】

控制面（control plane）负责「创建/删除/调度」等管理决策；数据面（data plane）负责实际跑流量的执行路径。分离的好处是控制面挂掉时数据面可继续服务——AENV 的 Scheduler HA 正是只保数据面路由（见第十一章）。

Insight#1 节点服务用单进程多任务而非多进程，换来的是内存态状态机与 CAS 协调，免掉一切外部协调依赖。

Insight#2 「编排器 + 内存 store + 文件持久化」的组合（service.rs L163–177）说明 AENV 把内存当事实源，磁盘只做崩溃恢复的兜底。

5.2 服务启动序列

main 函数里的初始化次序被严格设计过，理解这个次序等于理解各子系统的依赖关系。按顺序（行号见 src/bin/server.rs 及对应文件）：

- 内存分配器调优（L19–30）：全局使用 jemalloc，并用 malloc_conf 把 dirty/muzzy 页回收从默认 10 秒调成 1 秒——注释明说针对「RocksDB 打开、镜像解析、模板构建」这类突发分配后 RSS 不回落的问题，这是为大规模 RL 训练做的内存调优。

```
#[global_allocator]
static ALLOC: tikv_jemallocator::Jemalloc = tikv_jemallocator::Jemalloc;
// purge dirty/muzzy pages after 1s instead of the default 10s ...
#[export_name = "malloc_conf"]
pub static malloc_conf: &[u8] =
    b"dirty_decay_ms:1000,muzzy_decay_ms:1000,background_thread:true\0";
// src/bin/server.rs L19–30
```

- 日志与 Prometheus recorder 初始化（L58–59）。
- ConfigManager 全局单例（cfg.rs L1284–1296）：OnceLock，默认配置文件是编译期拼接的 config/default.toml；未初始化就调用 global() 会直接 panic（cfg.rs L1276–1281），所以必须最早完成。
- 权限收窄（L81–82）：先 require_runtime_capabilities 验证必需 Linux capabilities 存在，再 clear_ambient_capabilities 清掉 ambient set——最小权限原则。
- ApiKey::resolve（L84）与 NodeIdentity::from_config（L87）：node_id 默认取 hostname，commit 是编译期注入，故意不受运行时环境变量影响。
- P2P 传输层（L88–92）：基于 iroh，随后接入镜像缓存与 overlaybd P2P runtime。
- UblkDeviceManager 全局初始化（L97–101）：overlaybd 块设备的命脉。
- FirecrackerPool::prime(10s)（L103–105）：预热进程池；失败只 warn 不退出——warm pool 是优化而非必需，冷启动路径仍在。
- 核心四件套组装为 Arc（L111–127）：SnapshotManager、TemplateBuilder（携带集群 CPU 交集的 Arc>>）、ImageResolver、VolumeManager。
- Orchestrator 构造（L128–133）：内存元数据 store + 文件持久化 persister，构造时从磁盘恢复上一轮的 paused 沙箱（persister.load_all，service.rs L214–233）。
- Apilimpl 聚合所有句柄并 recover_image_builds 恢复中断的镜像构建（失败仅 warn）。
- server::new 构建 axum Router，绑定 0.0.0.0:8000，每连接 set_nodelay(true)（L183–187）。

停机顺序是启动的镜像反转，且被显式设计过：先 drain 快照 startup-manifest 任务（15 秒上限，宁可等也不让 manifest 丢失）→ abort 构建清理 → 停上报 reporter → orchestrator.shutdown() → FirecrackerPool → ublk daemon → overlaybd p2p → p2p transport（L191–230）。为什么不并行停机？因为依赖链有方向：编排器先停才能确保没有新任

务引用 Firecracker 池与 ublk 设备，反序 drain 才能保证「先没人用、再关资源」。

【零基础知识补给站】

jemalloc 是 FreeBSD 起源的内存分配器，比 glibc malloc 更擅长多线程与内存归还。「RSS 不回落」指进程常驻内存（Resident Set Size）在突发分配后不还回操作系统——通过 decay 参数让 jemalloc 更激进地归还页面。

Insight#1 启动序列里每一处「失败只 warn」（warm pool 预热、镜像构建恢复）都在说：服务可用性优先于单次优化，降级启动好过拒绝启动。

Insight#2 OnceLock

全局单例是最脆弱处之一：初始化顺序敏感，任何新的早期调用点都可能引入启动期 panic；测试代码里已出现 #[cfg(test)] 特判（cfg.rs L1299–1308）。

5.3 沙箱生命周期状态机

沙箱状态定义在 src/orchestrator/types.rs L74–84，共 8 个状态：

```
pub enum SandboxState {
    Creating, Resuming, Running, Snapshotting,
    Forking, Pausing, Paused, Killing,
}
// src/orchestrator/types.rs L74–84
```

其中 Running 与 Paused 是稳定态，Killing 是删除占位，其余都是过渡态。对外 API 只暴露 Running/Paused 两态——4 个过渡态被映射成 Paused（impls/sandbox.rs L79–89），外部观察不到内部转换细节。主要状态迁移如下：

起点	迁移到	触发	迁移方式
(无)	Creating	POST /sandboxes	新增元数据即 Creating
Creating	Running	envd 就绪 + init 完成	CAS update_if_state
Running	Pausing	pause 请求 / TTL 驱逐	CAS；并发 pause 汇入等待
Pausing	Paused	快照持久化成功	CAS；失败则两段降级回滚
Paused	Resuming	resume / connect / auto-resume	CAS；冲突时 join_concurrent_resume
Resuming	Running	从快照恢复完成	CAS；删除持久化记录
Running/Paused	Killing	DELETE / TTL Delete	CAS 重试循环，遇过渡态先等待
Killing	(删除)	Capture→Stop→Release 完成	DeleteProgress 状态机驱动

表 5-2 沙箱状态迁移表（src/orchestrator/service.rs）

所有迁移都是 CAS（Compare-And-Swap，比较并交换）：update_state_if_state 要求当前状态在 expected_states 集合内才迁移，失败返回

StateConflict。并发请求不排队重试，而是汇入——通过每沙箱一个 watch channel 等待同一个状态机的终态，再把终态翻译回各自的响应（join_concurrent_pause/resume, service.rs L2202–2231、L2236）。外层包 60 秒的 WAIT_TRANSITION_TIMEOUT 防止过渡态任务 panic 后等待者永久阻塞。

TTL 驱逐采用「认领式」设计：没有 per-sandbox 定时器，而是周期任务 start_auto_evict_task

每秒扫描过期集合，对每个过期沙箱先取删除进度锁（与显式删除互斥），再 CAS 认领为 Pausing 或 Killing，认领瞬间还要二次校验 is_expired——处理「扫描到过期但刚被 keep-alive 续期」的竞态（service.rs L2266–2315、L1183–1199）。删除则用 DeleteProgress 状态机（Capture→Stop→Release→Done）实现幂等：重复 DELETE 汇入同一个进度对象，Done 直接返回 Ok。

【零基础知识补给站】

CAS 是原子操作：只有当当前值等于预期值时才写入新值。把状态迁移全部做成 CAS，多个并发请求中只有一个能「认领」迁移权，其余的要么失败要么等待，从根本上杜绝了「两个 pause 同时干活」的竞态。

Insight#1 把 4 个过渡态对外映射成 Paused

是信息隐藏的好例子：外部消费者不需要理解内部状态机，内部重构也不破坏 API 契约。

Insight#2 WAIT_TRANSITION_TIMEOUT=60s 是防止过渡态卡死的唯一保险——若某操作持 Pausing/Killing 状态 panic

且未回滚，该沙箱会永久卡死直到重启，这是状态机最脆弱处（service.rs L60–63 注释自承）。

5.4 create 调用链逐步拆解（热路径）

POST /sandboxes 的热创建路径（从已提交快照启动）是全系统最频繁的路径，逐步拆解如下（括号为文件:行号）：

```

Apilmpl::sandboxes_post      src/api/impls/sandbox.rs:1008
→ SnapshotManager::load_runnable(template_id) // 模板→可运行快照
→ Orchestrator::create_sandbox      src/orchestrator/service.rs:424
→ run_cancellation_safe("create", ...) // spawn 独立 task，客户端取消不影响执行
→ create_sandbox_inner              service.rs:453
→ 校验 virtualization_mode 匹配 (L486–493)
→ secure 沙箱生成 envd_access_token (L478)
→ launch_sandbox(LaunchPlan::for_create_from_snapshot ...) service.rs:2400
→ FirecrackerSandboxFactory::build_from_snapshot factory.rs:146
→ protect_image_refs // 钉住镜像层，防 GC 误删
→ sandbox.start_nowait()            sandbox.rs:877
→ handle 注册 + store.add(过渡态) // 此时才对外可见
→ sandbox.wait_for_ready()          // 轮询 envd 就绪
→ store.update_if_state(→ Running) + upsert_proxy_route

```

launch_sandbox 的失败处理按 FailedLaunchStage 分三档（Registered / TransitionalPersisted / RunningPersisted, service.rs L91–110），决定回滚到哪个状态、

是否摘代理路由；detach_launch_runtime_if_current (L2664) 用 Arc::ptr_eq 防止清理掉并发替换的新 handle——这是并发同名沙箱替换的关键细节。

start_fresh (sandbox.rs L1781–1960) 是真正把 Firecracker

进程拉起来的链路：link_tools_drive → UblkDeviceManager 为 rootfs 创建 overlaybd 运行时块设备 → 拼 boot args (补 init=/init 与额外盘参数) → NetworkManager 分配网络 slot (netns + tap + IP) → fc_instance.spawn_with_netns 在沙箱网络命名空间内启动 Firecracker (它直接能看到 tap0) → 等 API socket → configure_microvm → InstanceStart → envd 就绪探测。

```
// —— Spawn Firecracker inside the network namespace so it can access tap0 ——
self.fc_instance.spawn_with_netns(&firecracker_binary, stdout_path.as_deref(),
                                   stderr_path.as_deref(), Some(&netns)).await?;
// src/sandbox/firecracker/sandbox.rs L1926–1937
```

为什么不先起进程再搬进 netns？因为「在 netns 里

spawn」让网络隔离在进程创建瞬间即生效，没有中间窗口期，也无需后续搬移系统调用。

【零基础知识补给站】

netns (network namespace) 是 Linux 的网络隔离单元，每个 netns 有独立的网卡、IP、路由表。tap0 是虚拟机用的虚拟网卡。把 Firecracker 放进沙箱 netns，等于给它一台「只看得见自己网络」的独立机器。

Insight#1 「store.add 在 handle 注册之后、wait_for_ready 之前」的顺序意味着沙箱对外可见时仍是过渡态，状态机而非「先跑起来再登记」保证了可观测性与并发正确性。

Insight#2 三档 FailedLaunchStage

的本质是「故障会计」：每一步失败都只清理已经创建的东西，绝不多删、也不少删。

5.5 并发与可靠性模型

AENV 并发模型的精髓是锁分层。五层结构各司其职：

层	结构	锁与机制
元数据	InMemoryMetadataStore	单 RwLock；每沙箱 watch channel；notify 在释放写锁后
运行时句柄	sandboxes: RwLock>	map 锁只用于查找/插拔；每沙箱一把 tokio Mutex 串行化 backend 操作
删除	deletions: Mutex>	DeleteProgress 对象再套 Mutex，多调用方汇入同一删除
代理路由	proxy_routes: RwLock + AtomicU64 版本号	单调递增版本号让读方检测路由过期
事件	broadcast::Sender (容量 1024)	无锁广播给观测层

表 5-3 锁分层 (src/orchestrator/service.rs L120–124、store/in_memory.rs)

跨子系统操作的统一纪律是：CAS 状态 → 摘下/挂上 handle → 慢操作（持单沙箱 Mutex）→ CAS 落定。慢操作期间元数据处于过渡态，其他操作要么冲突失败要么 wait_for_transition。所有生命周期操作外面还套着 run_cancellation_safe（service.rs L288-314）：

```
let (tx, rx) = oneshot::channel();
tokio::spawn(async move {
    let result = future.await;
    if tx.send(result).is_err() { /* caller stopped waiting */ }
});
rx.await...
// src/orchestrator/service.rs L288-314
```

即使 HTTP 客户端断开导致外层 future 被 drop，实际操作仍在独立 spawn 的 task 里跑完——不会因为客户端取消而在 VM 启动到一半时留下孤儿 Firecracker 进程。代价是调用方无法真正取消操作，只能放弃等待。为什么不直接让 future 可取消？因为 VM 启动是「已分配 netns、ublk 设备、镜像钉」的多资源事务，中途取消的回滚复杂度远高于「跑完再清理」；用 spawn+oneshot 把「调用方可取消」与「操作必须跑完」解耦，是复杂度最低的正确性方案。

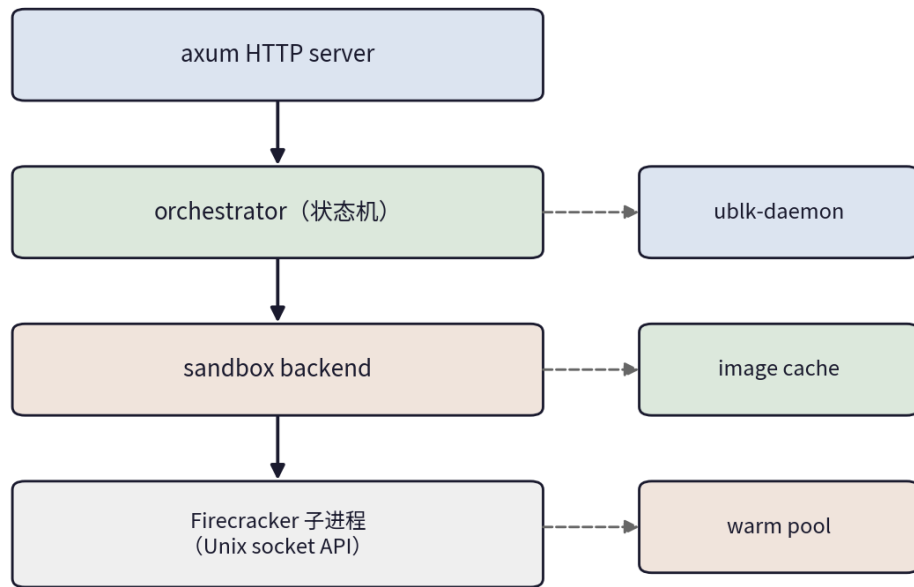
这套「元数据 CAS + watch 过渡态 + run_cancellation_safe」三件套叠加 DeleteProgress 幂等删除，让一个无外部协调器（无 etcd/DB 锁）的单机控制面在高并发 RL 负载下保持自治——这是整个控制面并发正确性的基石。

【零基础知识补给站】

tokio 是 Rust 的异步运行时；Mutex 保证同一时刻只有一个任务操作某个沙箱；watch channel 是「最新值广播」原语，适合状态通知——等待者订阅后，状态一变就收到，且 notify 被刻意安排在写锁释放之后，避免等待者醒来又撞上锁。

Insight#1 后台周期任务用 Arc::downgrade + upgrade 持有 Orchestrator（service.rs L2329/2344），进程退出时任务自然终止——Weak 自引用是「后台任务不阻止关停」的优雅写法。

Insight#2 每沙箱一把 Mutex 而非全局大锁，让不同沙箱的 pause/resume 完全并行，锁粒度与业务实体一一对应。



控制面自上而下逐层委派；右侧为旁挂的块设备、镜像缓存与预热池组件

图 5-1 AENV 三层架构与沙箱生命周期状态机（示意）

第六章 隔离层：为什么是 Firecracker microVM

前面几章我们看到，AgentENV 把「环境」变成了一台可以随时暂停、快照、分叉的微型虚拟机。但一个更根本的问题还没有回答：为什么隔离层非要是虚拟机？容器（Container）不是更快、更轻吗？本章我们从「隔离光谱」讲起，逐步推到 AgentENV 的选择——Firecracker microVM，并拆解 AgentENV 驱动它的方式（二进制子进程 + Unix Socket API）、两处关键定制补丁，以及每个沙箱一套独立网络命名空间的隔离方案。读完本章，你会理解：隔离性、速度、可快照性是一个三元权衡，而 AgentENV 的答案是把三者各自推到极限后再组合。

6.1 隔离光谱：从共享内核到独立内核

所谓「隔离」，回答的是一个安全问题：当沙箱里跑的代码是 AI Agent 自动生成的、可能携带恶意或破坏性逻辑时，它能伤害到谁？业界的隔离方案落在一条光谱上，我们用三个代表性位置来刻画它。

方案	内核归属	攻击面	启动开销	典型代表
容器（Container）	与宿主共享同一个 Linux 内核	整个宿主内核的系统调用接口；Agent 拿到 root 后直面内核	毫秒级	Docker / containerd
microVM（微型虚拟机）	每个沙箱独立内核（KVM 虚拟化）	极小的设备模型 + KVM 接口	百毫秒级（可再优化）	Firecracker
全功能虚拟机（Full VM）	独立内核	完整 QEMU 设备模型，代码量巨大	秒级	QEMU / VMware

容器的本质是「同一个内核上的进程分组」：namespace 负责「看不见别人」，cgroup 负责「用不多资源」，但所有容器共用一个内核。这意味着攻击面（Attack Surface，即攻击者能触碰到的代码面积）是整个内核的系统调用表——数百个系统调用、无数 ioctl，任何一处内核漏洞都可能被沙箱里的 root 进程用来逃逸。社区分析直言：「agent 带 root，内核共享面太大」^[13]。而 RL（Reinforcement Learning，强化学习）场景要同时跑成千上万个沙箱，Agent 生成的代码必然要执行任意 shell 命令——把赌注押在「内核没有漏洞」上是不可接受的。

光谱的另一端，全功能虚拟机用 QEMU 模拟一整台计算机：网卡、显卡、USB、软驱……设备模型代码量以百万行计，每一块都是攻击面，而且启动以秒计。「为什么不直接用 QEMU？」因为它为通用性付了太多税：AgentENV 只需要 CPU、内存、一块磁盘、一张网卡、一个串口，其余全是负担。microVM 的思路正是把设备模型裁到最小，只保留 virtio 标准设备，把攻击面和启动开销同时压下去^[5]。

【零基础知识补给站】

什么是 KVM？KVM（Kernel-based Virtual Machine，基于内核的虚拟机）是 Linux 内核内置的虚拟化模块，它把「创建虚拟机」变成了一组系统调用：打开 /dev/kvm，创建 VM、创建 vCPU、告诉内核哪段内存是虚拟机的物理内存，然后让 vCPU

跑起来。硬件的虚拟化扩展（Intel VT-x / AMD-V）保证虚拟机里的代码几乎以原生速度执行。KVM 本身不提供任何设备——设备模拟留给用户态程序去做，Firecracker 就是那个「用户态程序」^[7]。

Insight#1 隔离强度的本质是「攻击面大小」：容器共享内核，攻击面是整个内核；microVM 独立内核，攻击面收缩为 KVM 接口加一个小设备模型。

Insight#2 「为什么不

X？」是本章的基本方法论：不用容器是因为共享内核不够安全，不用全 VM 是因为 QEMU 太重——Firecracker 是光谱上被精确选中的点。

6.2 Firecracker 是什么：KVM 之上的最小 VMM

Firecracker 是 AWS 开源的 VMM（Virtual Machine Monitor，虚拟机监视器），用 Rust 编写，最初为 Lambda 无服务器计算设计。它的定位在 Firecracker 论文里说得很清楚：在安全隔离的前提下，把虚拟机的开销压到接近容器^[5]。

它的架构可以用四个关键词概括：

- **KVM 打底**：Firecracker 自己不做虚拟化的脏活，创建 VM、跑 vCPU、注入中断全部通过 /dev/kvm 交给内核^[7]。Firecracker 进程本质上是一个「KVM 配置器 + 设备模拟器」。
- **最小设备模型**：只实现 virtio-block（磁盘）、virtio-net（网卡）、串口、virtio-balloon（内存气球）等少数设备，砍掉 BIOS 改用直接加载内核。设备越少，攻击面越小，启动也越快。
- **Unix Socket API**：Firecracker 启动后在指定路径创建一个 Unix Domain Socket，宿主程序用 HTTP/JSON 风格的 REST API 控制它：PUT /machine-config 配 CPU 内存、PUT /drives/rootfs 挂盘、PUT /snapshot/create 做快照、PATCH /vm 做 pause/resume。所有控制都是「往一个本地 socket 发 HTTP 请求」，没有任何 SDK 耦合^[1]。
- **单进程单 VM**：一个 Firecracker 进程对应一台 microVM，内部线程极少（API 线程、vCPU 线程、I/O 线程），进程死了 VM 就没了——这让生命周期管理变得极其简单：杀进程即销毁。

这种设计对 AgentENV 的意义在于：microVM 的一切——配置、启动、暂停、快照、恢复——都被外化成了一组明确的 API 调用。AgentENV 不需要理解虚拟化内部，只需要成为一个优秀的「API 编排者」。这正是下一节的主题。

【零基础知识补给站】

什么是 virtio？virtio 是虚拟机与宿主之间的一套「半虚拟化」设备标准：guest 内核里装一个 virtio 驱动，宿主（VMM）实现对应的 virtio 设备，双方通过共享内存环形队列交换数据。它不走真实硬件的复杂协议，专为虚拟化设计，因此又快又简单。virtio-block 是磁盘、virtio-net 是网卡。

Insight#1 Firecracker 把「虚拟机」从一台模拟的计算机降维成「一个进程 + 一个 socket」，这是快照、分叉等上层魔法能成立的地基。

Insight#2 最小设备模型同时赢得安全性与速度：攻击面小、启动快，代价只是 guest 不能用花哨硬件——而沙箱本来就不需要。

6.3 AgentENV 如何驱动 Firecracker：子进程、netns 与 OOM 保护

一个容易忽略的架构事实是：AgentENV 没有把 Firecracker 当作库（library）链接进来，而是把它当作**二进制子进程**启动^[1]。封装类是 `src/sandbox/firecracker/instance.rs` 中的 `FirecrackerInstance`（第 34 行），它持有三件东西：`process: Option<Child>`（子进程句柄）、`socket_path`（API socket 路径）、`UnixSocketClient`（HTTP 客户端）。二进制路径默认是 `<home>/bin/firecracker`（`src/cfg.rs:750` 的 `resolved_firecracker_binary_path()`）。

「为什么不把 VMM 做成库链接进主进程？」排除法可以很清楚地回答：

- **故障隔离**：VMM 崩了只死一台沙箱；若 VMM 代码跑在主服务进程内，一个空指针就带走整台宿主机上的所有沙箱。
- **权限与命名空间边界**：子进程可以被单独放进网络命名空间、单独调 OOM 优先级，库做不到。
- **版本可替换**：换一个打过补丁的 Firecracker 二进制即可升级，不需要重新编译主服务。

启动入口是 `spawn_with_netns()`（`instance.rs:64`）。它做的三件事都值得细看。第一，`Command::new(firecracker_binary)` 带参数 `--api-sock <path>` 等，并设 `process_group(0)` 让子进程独立成组，便于整组信号管理。第二，也是最关键的一步——`spawn` 之前先进入目标网络命名空间：

```
let child = crate::privileges::spawn_tokio_command_scoped(cmd, &[], move || {
    if let Some(netns) = netns {
        let rc = unsafe { libc::setns(netns.as_raw_fd(), libc::CLONE_NEWNET) };
        if rc != 0 { return Err(std::io::Error::last_os_error()); }
    }
    Ok(())
}).await.context("failed to spawn firecracker");
// src/sandbox/firecracker/instance.rs:120-130 [1]
```

`setns(fd, CLONE_NEWNET)` 让即将诞生的 Firecracker 进程**出生就在沙箱的网络命名空间里**，这样它一睁眼就能看到本沙箱的 `tap0` 网卡并把 `virtio-net` 接上去。「为什么不先 `spawn` 再把设备搬进 `netns`？」因为网络设备跨命名空间搬移有时序窗口和竞态，而「在正确的命名空间里出生」是原子生效的——网络隔离在进程创建瞬间即成立^[1]。

第三件事是 `spawn` 之后立刻写 OOM 优先级：

```
// oom_score_adj = 1000: 让 OOM killer 先杀 firecracker, 保护 agentenv 主服务
if let Some(pid) = child.id() {
    let oom_path = format!("/proc/{pid}/oom_score_adj");
```

```
if let Err(e) = fs::write(&oom_path, b"1000") { warn!(...) }
}
—— src/sandbox/firecracker/instance.rs:137-142 [1]
```

oom_score_adj 是 Linux OOM killer（内存耗尽时的杀手）的打分偏移，范围 -1000 到 1000，分数越高越先被杀。把 Firecracker 设成 1000（最高），等于宣告：「内存不够时请先牺牲沙箱，绝不牺牲编排主服务」。这是多租户系统的自保本能——主服务死了所有沙箱都成孤儿，而一个沙箱死了只是局部损失。

控制通道方面，AgentENV 写了

UnixSocketClient（src/sandbox/firecracker/socket.rs）：基于 hyper 的 legacy client 加一个自定义 UnixConnector，URI 形如 http://localhost/machine-config，body 是 JSON。上层的请求/响应模型则由 thirdparty/firecracker-client/ 提供——那是用 OpenAPI Generator 7.22.0 从 firecracker.yaml 生成的 Rust 客户端（API version 1.15.1）^[1]。「为什么不自己手写模型？」因为 Firecracker API 字段繁多且随版本演进，用 schema 生成客户端能保证字段名、类型与上游严格同步；而生成的 schema 文件本身，恰是下一节「打补丁」的落点。

【零基础知识补给站】

什么是网络命名空间（netns）？Linux namespace 的一种，每个 netns 有自己独立的网卡、IP 地址、路由表和防火墙规则。两个 netns 里的进程默认完全看不见对方的网络。setns() 系统调用可以让一个进程「走进」某个已有的命名空间。

【零基础知识补给站】

什么是 Unix Domain

Socket？一种只在本机有效的「文件型」socket：通信双方通过文件系统里的一个路径（如 /run/aenv/fc.sock）建立连接，不走网卡、不经 TCP 协议栈，天然免疫外部网络攻击，开销也更低。在它上面照样可以讲 HTTP 协议。

Insight#1 「二进制子进程而非库」是整套系统的故障隔离基石：VMM 崩溃 = 单沙箱死亡，主服务永远活着。

Insight#2 setns 先于 spawn 是一个教科书级的技巧：让进程「出生在正确的命名空间」，比「出生后再搬进去」原子、无竞态。

Insight#3 oom_score_adj=1000

用一行写入表达了清晰的优先级：编排者的命比沙箱的命值钱。

6.4 定制补丁：上游 Firecracker 没有的两个端点

AgentENV 使用的 Firecracker 不是原版。thirdparty/firecracker-client/ 里的 firecracker.yaml 被打过补丁，包含两个上游没有的 API^[1]：

- **GET /vm/dirty-memory-ranges**（firecracker.yaml:856 起）：返回 DirtyMemoryRanges { page_size, memory_size, ranges: [{base_host_virt_addr, image_offset, length}] }。它把 KVM 脏页跟踪的结果翻译成「宿主虚拟地址 +

内存镜像偏移 + 长度」的三元组列表，供外部进程用 `process_vm_readv()` 直接读 guest 内存。关键语义是**逻辑上非清除 (non-clearing)**：读完后脏位保留，快照打包任何一步失败都能安全重试，脏页不会因为一次失败查询而丢失^[1]。

- **SnapshotCreateParams.mem_file_path 改为 Option<String>**：上游该字段必填，即快照必然包含全量内存；改成可选后，不指定就只存 `vm_state`（设备与 vCPU 状态），不存内存——这打开了「只存 `vm_state` 的 diff 快照」这条路，内存部分交给自定义链路处理^[1]。

这两个补丁意味着什么？上游 Firecracker

的快照模型是「一把梭」：内存整体落盘成一个文件。这对 Lambda 够用，但对 AgentENV 想要的「增量快照 + 分叉」不够——每次快照都写全量内存既慢又大。两个补丁合在一起，等于把快照的「内存部分」从 Firecracker 手里夺过来：Firecracker 只负责回答「哪些页脏了」，把脏页读出来、写成增量层、分层管理、上传到 S3——全部由 AgentENV 自己实现（这是第九章的主角）。可以说，**AgentENV 对 Firecracker 的定制不是功能扩展，而是把快照的主导权上移到了编排层。**

代价同样明显：整个内存快照链路依赖打过补丁的 Firecracker，而仓库里 `thirdparty/` 只 vendored 了生成的客户端代码，未见补丁源码随仓库管理^[1]。一旦升级上游 Firecracker 而补丁未跟随，`pause/fork` 全链路有静默退化的风险。这是技术债，也是第九章 9.6 节要讨论的脆弱面之一。

【零基础知识补给站】

什么是脏页跟踪 (dirty page tracking)？KVM 可以记录虚拟机运行期间「哪些内存页被写过」，写过的页叫脏页。做增量快照时只需保存脏页而非全部内存。AgentENV 通过 `set_machine_config(..., track_dirty_pages=true)` (`instance.rs:276`) 开启它，且要求 `page_size` 必须为 4096、区间 4K 对齐^[1]。

Insight#1 两个补丁的共同指向：把「内存快照」从 Firecracker 的内置功能变成编排层的自建链路，增量快照与 `fork` 才有可能。

Insight#2 非清除语义是失败恢复的保险丝：查询脏页不清脏位，打包失败重试时数据还在——好补丁不仅要加功能，还要设计好失败语义。

6.5 网络隔离：每沙箱一个 netns、一对 veth、一条默认拒绝的出口

虚拟机隔离了内核，但网络还需要单独回答：成千上万个沙箱在同一台宿主机上，如何保证它们的流量互不串扰、且不能随意访问内网？AgentENV 的答案是三层结构^[1]。

第一层：地址池与 slot。NetworkManager 维护两块地址池（默认 10.11.0.0/16 与 10.12.0.0/16）和一组预分配的 slot。每个沙箱启动时经 `NetworkManager::global().allocate_any()` 拿到一个独立 Slot (`src/sandbox/network/slot.rs:54`)，Slot 是 RAI 资源：存活期间独占一个 netns 与一对 IP，drop 时自动归还。「为什么用预分配 slot 而不是现用现建？」因为建 netns、建 veth、配路由是一串 netlink

操作，热路径上做一次要几毫秒；池化之后fork/恢复路径直接拿现成的，这也是预热池三元组 WarmFirecracker { slot, fc_instance, work_dir } 里 slot 占第一位的原因^[1]。

第二层：veth 对与 tap0。每个 slot 在宿主侧有一个 veth-`<idx>` 接口（HOST_VETH_PREFIX = "veth-", manager.rs:23），它是 veth pair（虚拟以太网对，一根虚拟网线的两端）的宿主端，另一端伸进沙箱 netns；沙箱 netns 内再建 tap0 交给 Firecracker 的 virtio-net。数据路径是：guest → tap0 →（netns 内路由）→ veth 对 → 宿主协议栈。Firecracker 进程本身也住在这个 netns 里（6.3 节的 setns），所以它开 tap0 不需要任何跨命名空间操作。

第三层：egress 策略默认拒绝内网。出口策略由 src/sandbox/network/policy.rs 实现，配置项 [network.egress] always_denied_cidrs 默认拒绝 RFC1918 私有地址段（10.0.0.0/8、172.16.0.0/12、192.168.0.0/16）与链路本地地址^[1]。「为什么默认拒绝 RFC1918？」因为宿主大概率身处某个内网：数据库、元数据服务、同事的开发机都在 RFC1918 段里。沙箱里跑的是不可信代码，默认放通内网等于把内网暴露给所有 Agent。默认拒绝、按需放通，是「最小权限原则」在网络层的落地。PUT /sandboxes/{id}/network 可以整体替换某个沙箱的出口策略，实现 per-sandbox 的精细管控^[1]。

【零基础知识补给站】

什么是 veth pair？virtual ethernet pair，内核里一对成对出现的虚拟网卡，从一端发出的包会从另一端收到，像一根虚拟网线。它常用来连接两个网络命名空间，或连接 netns 与宿主。tap 设备则是另一种虚拟网卡，工作在二层，常被虚拟机用来收发原始以太网帧。

【零基础知识补给站】

什么是 RFC1918？一份互联网标准文档，规定了三段「私有 IP 地址」（10.0.0.0/8、172.16.0.0/12、192.168.0.0/16），这些地址不会在公网路由，专供内网使用。公司内网、家庭路由器背后几乎都是这些地址。

Insight#1 网络隔离的完整闭环：netns

负责「互相看不见」，地址池负责「规模化分配」，egress 默认拒绝负责「看不见内网」。

Insight#2 slot 池化把「建网络」从沙箱启动的热路径上移除了——隔离性与速度在这里不矛盾，靠的是预分配。



隔离强度递增、开销也递增：microVM 在安全与轻量之间取得平衡

图 6-1 AgentENV 隔离层总览：宿主内核之上，每个沙箱 = 独立 netns + Firecracker 子进程 + microVM 独立内核

第七章 存储与镜像：OverlayBD、缓存与分发

前面几章反复出现一个论断：AgentENV

能把环境启动压到几十毫秒，靠的不是「启动得更快」，而是「根本不启动」——快照恢复 + 镜像按需加载。本章把镜头对准这条论断的地基：存储与镜像子系统。我们将回答一个看似不可能的问题：**一个集群可用的镜像集合，凭什么能超过单块磁盘容量好几个数量级？**答案由三层构成：OverlayBD 的层模型让镜像变成可远程寻址的索引（7.2-7.3），有界缓存让本地磁盘只做热数据的「夹层」（7.4），P2P 分发让热点层在节点之间横向流动（7.6）。中间穿插 OCI 生态的兼容与转换（7.5）。

7.1 问题：镜像集合为什么能超过单盘容量几个数量级

传统容器平台的隐含假设是「镜像是少数几个、被反复使用的」——一个集群常驻几十到几百个镜像，每台节点把用到的镜像 docker pull 下来即可。Agentic RL（Agent 强化学习，让模型在真实软件环境里反复试错学习）打破了这个假设：不同的训练任务需要不同的操作系统、语言运行时、编译器、依赖版本、工具链与代码仓库组合，「随着任务数增长，所需环境很快变成一个庞大且高度多样化的镜像集合，其总大小可能远超单节点的存储容量」^[2]。生产侧的口径是：K3 技术报告提到 AgentENV 支撑了 150 万镜像的规模^[3]；社区实测文章也引用了同一数字^[16]。

150 万个镜像是什么概念？即便平均每个压缩后只有 200 MB，全量约 300 TB，而单节点本地盘通常只有几 TB；「把全部镜像预热到每台节点」在带宽与时间上也完全不可行——环境就位要求毫秒级，等不起 GB 级的全量下载。**为什么不用共享文件系统挂镜像？**随机小读延迟与元数据开销会在几千并发下压垮它，且 tar 归档不支持高效随机块寻址。**为什么不「调度时先拉镜像」？**环境就位延迟直接计入训练吞吐，环境慢一秒，GPU 空转一秒。**为什么不用容器共享内核省镜像？**共享内核省内存不省镜像，且 reward 驱动的 agent 会主动尝试突破边界，共享内核攻击面不可接受^[2]。

排除法走完，剩下的路只有一条：**镜像不能只以「文件」的形式存在，必须变成「可按需寻址的块设备」**——启动环境时只读索引，运行中缺哪一块才取哪一块，本地磁盘降级为缓存。这正是 OverlayBD（overlay block device，阿里云与蚂蚁开源的按需加载块设备镜像格式）的设计原点^[6]，也是 AgentENV 存储子系统的底座。

【零基础知识补给站】

镜像（Image）：把一个环境需要的操作系统、运行时、代码打包成的只读模板，类似「系统安装光盘」。

块设备（Block Device）：操作系统眼里「可以按 512 字节扇区随机读写」的设备，硬盘就是最典型的块设备。文件系统（ext4 等）就建在块设备之上。

OCI（Open Container Initiative）：容器镜像与运行时的事实标准组织，Docker 镜像就是 OCI 镜像的一种^[24]。

按需加载（On-demand Loading / Lazy Loading）：启动时只下载「目录」（索引），真正读

某块数据时才下载那块，类似视频网站的「边下边播」。

Insight#1 镜像爆炸是 Agentic RL 特有的负载特征：环境多样性随任务组合数增长，而不是随并发数增长——这决定了存储子系统必须为「长尾镜像」设计，而不是为「热点镜像」设计。

Insight#2 「磁盘从存储介质降级为缓存」是本章最重要的观念转换：一旦接受这个设定，容量问题就转化成了命中率与驱逐策略问题。

7.2 OverlayBD 层模型：LSMT、16 字节段映射与 390 字节头

OverlayBD 把镜像层（layer）表达为 **LSMT (Log-Structured**

Merge-Tree, 日志结构合并树) 格式：数据追加写入，配一份「逻辑偏移 →

物理偏移」索引^[6]。AgentENV 在 storage/overlaybd/ 用 Rust

重新实现了这套格式，理解它只需两个结构体。第一个是

SegmentMapping (段映射)：索引里的每条记录，回答「虚拟盘第 X

段扇区的真实数据在文件第 Y 段扇区」。磁盘编码被压到 **恰好 16 字节** (format.rs

L13-44)：

```
// storage/overlaybd/src/lsmt/format.rs:13-44 (磁盘编码, #[repr(C, packed)])
pub struct DiskSegmentMapping { data_low: U64, data_high: U64 }
// low u64 = offset(50 bit) | length(14 bit) // 目的偏移 / 段长, 单位 512B 扇区
// high u64 = moffset(55 bit) | zeroed(1) | tag(8) // 物理偏移 / 全零标记 / 标签
const _: () = assert!(std::mem::size_of::<DiskSegmentMapping>() == 16); // 编译期断言
```

50 bit 扇区偏移意味着单盘可寻址 $2^{50} \times 512 \text{ B} = 512 \text{ PiB}$ 虚拟空间；**zeroed**

位则把稀疏性编码进了格式本身：一段全零数据不需要任何物理存储，打个标记即可。全 1

的物理偏移保留为 NO_PHYSICAL_OFFSET 哨兵 (format.rs L9)。第二个结构是

HeaderTrailer (头/尾标)：每个 LSMT 文件首尾各一份 390 字节元数据 (format.rs

L78-102, 编译期断言 size_of == 390)，含魔数

"LSMT\0\1\2"、索引偏移与长度、虚拟盘大小、本层与父层 UUID (各 37 字节)、256 字节

user_tag。头尾双写是为了崩溃恢复：写一半断电，至少能从另一端判断文件不完整。flags

区分 header/trailer、data/index，以及关键状态位 **sealed (已封存)** 与 sparse/hybrid

写模式。

sealed 层与 upper 层的分工是核心心智模型：只读层 (lower) 写完即 set_sealed 封存、永

不修改，因此可被任意多镜像与沙箱安全共享；每个运行中沙箱有且仅有一个可写顶层 (uppe

r) 承接所有写。快照 = 封存当前 upper 再开新 upper——guest 的磁盘写在运行时本就累积在

upper 里，快照时 seal

即可，无需重新发现与拷贝，这就是磁盘侧快照几乎零成本的格式基础^[13]。

层数有硬上限 **MAX_STACK_LAYERS = 255** (lsmt/file/types.rs

L20)；内存快照压实把预算动态定在 32 层以内 (预算 = 32 - 稳定前缀层数/4，至少

1；overlaybd_snapshot.rs L40、L416 附近)，远离硬上限^[1]。可写 upper 有三种

UpperMode (config.rs L47)：**Sparse** (原地覆写稀疏文件，最简单但无法增量封存)、**Lo**

gStructured (所有写追加日志尾部，天然可封存，writable_mode

缺省)、HybridLogStructured (热数据原地写、封存时转日志, ublk 运行时默认)^[1]。为什么不只用 Sparse? 原地覆写让封存变成昂贵的全量拷贝; 日志结构让封存退化为一次元数据标记——快照廉价的前提。

【零基础知识补给站】

LSMT (日志结构合并树): 一种「只追加、不覆写」的存储格式, 所有修改都写在末尾, 定期合并。好处是写入快、崩溃恢复简单、历史版本天然保留; 代价是读要查索引、空间要定期压实。

扇区 (Sector): 块设备的最小寻址单位, 惯例 512 字节。SegmentMapping 里所有偏移都以扇区计。

UUID / 父层 UUID: 每一层的全球唯一编号和它「爸爸」的编号, 由此把一串层文件连成一条链——这就是「镜像分层」在磁盘上的真实形态。

编译期断言 (const assert): 让编译器在编译时检查「这个结构体必须恰好 16/390 字节」, 防止有人改结构体悄悄破坏磁盘格式兼容性。

Insight#1 OverlayBD 的格式设计把「稀疏性」和「可封存性」编码进了 16 字节的映射记录里——性能不是优化出来的, 是被数据结构设计出来的。

Insight#2 255 层上限 + 32 层预算的「双层护栏」值得学习: 硬限制由格式保证, 软预算由策略动态收紧, 两者之间的空间就是安全余量。

7.3 按需加载: 一次 guest 读请求的完整旅程

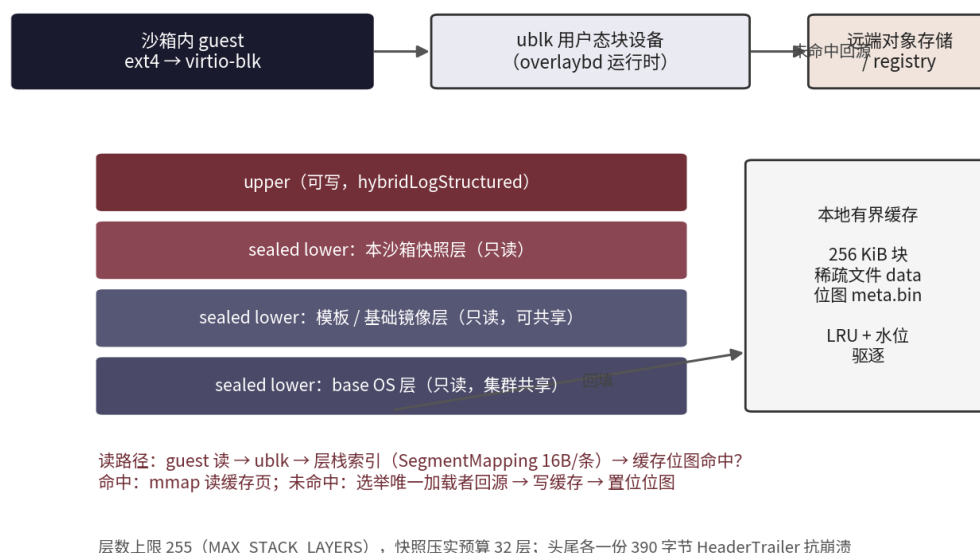


图 7-1 OverlayBD 层栈与按需加载: 沙箱只写最顶层的 upper, 只读 lowers 可远程、可共享

现在把镜头拉到运行时。一个沙箱启动时, AgentENV

只打开各层的索引, 不下载任何数据——open_lower_layer /

open_ro_remote (storage/overlaybd/src/image/image_file.rs

L550、L702) 拿到的可能只是远端 URL 与一份 LSMT 索引。索引很小 (16 字节 ×

段数)，一个 1 GB 的层以 256 KiB 块计只有 4096 条映射。从此刻起，guest 里的每一次读都可能触发如下旅程：

- ① **guest 读**：VM 内读文件 → ext4 → virtio-blk → host 侧 ublk 设备（ublk 是 Linux 用户态块设备框架，块设备逻辑跑在用户进程，便于迭代与隔离^[9]）。
- ② **索引查询**：沿层栈查 LSMT 索引，定位到某层的 256 KiB 块（DEFAULT_BLOCK_SIZE，cache/mod.rs）。
- ③ **位图中检查**：本地缓存为每个远端层维护稀疏文件 data 与位图 meta.bin（魔数 "OBCH"），命中则直接 mmap 读缓存页。
- ④ **未命中回填**：选举协议保证全进程只有一个「当选加载者」回源拉这一块，其余读者等待。
- ⑤ **落缓存置位**：写进稀疏文件、置位位图，后续读即命中。

回填写入这段代码值得逐行读，因为它示范了「稀疏文件 + 预分配 + mmap」如何把磁盘耗尽从灾难降级为普通错误（storage/overlaybd/src/backend/cache/full_file_cache/cache_entry.rs L465-487）：

```
// storage/overlaybd/src/backend/cache/full_file_cache/cache_entry.rs:465-487
pub(crate) fn write_block(&self, block_id: u64, data: &[u8]) -> Result<()> {
    let offset = block_id.saturating_mul(self.block_size);
    self.ensure_range_allocated(offset, data.len() as u64)?; // 先 fallocate 预占磁盘块
    let buf = unsafe { self.mem_region.get_mut(offset, data.len() as u64)
        .ok_or_else(|| anyhow!("mmap get_mut out of range ..."))? };
    buf.copy_from_slice(data); // 经 mmap 直接写入缓存文件
    self.mark_block_cached(block_id); // 最后才置位位图：位图永远落后于数据，绝不超前
    Ok(())
}
```

为什么要 ensure_range_allocated 预分配？注释写得很直白：直接往共享 mmap 的空洞里写，内核在缺页时才分配磁盘块，若磁盘满了，进程收到的不是可处理的 ENOSPC 错误而是 **SIGBUS 信号**——不可恢复。预分配把「磁盘满」降级为回填路径上的普通错误，读请求随后回退直连源站（cache_entry.rs L398-414

注释）。**为什么不能先置位再写数据？**那样并发读者会按位图读到未写完的脏页。「数据 → 位图」的发布顺序与下一节驱逐的「位图 → 洞穿」构成一对不变量：**位图声明过的块，数据一定在**。前台与后台之间还有一道闸门：download_gate 的 FgReadGuard（image_file.rs L793 起）给前台读更高 I/O 优先级，防后台预取饿死 guest 实时读；后台下载块 16 MiB、并发 4^[1]。

【零基础知识补给站】

ublk：Linux 5.15+ 引入的用户态块设备框架。传统上写块设备驱动要写内核模块，ublk 让你用普通用户进程实现块设备逻辑，内核通过共享内存环（io_uring）把请求转给你。崩了只崩进程，不崩内核^[9]。

mmap：把文件映射进内存地址空间，读写内存即读写文件，由内核自动换入换出。

SIGBUS：一种致命的进程信号，访问了「映射了但底层不存在」的内存时触发——对服务进程来说基本等于崩溃。

位图 (Bitmap)：每个块 1 bit 的「有没有」清单，4096 个块只需 512 字节。

Insight#1 「读路径上每一跳都是可选的」是按需加载的灵魂：命中位图就走本地，没命中才回源，回源失败还能直连——性能是逐层 degradable（可降级）的。

Insight#2 先写数据后置位图、先预分配再写 mmap，这两个顺序都是「把不可恢复的硬件/内核故障翻译成可处理的应用错误」——存储工程的功夫大半在这种顺序里。

7.4 有界缓存与驱逐：块级 FileCache 与引用图 GC 的双层结构

「本地盘只是缓存」意味着它必须有界且可驱逐。AgentENV 的驱逐是**两层结构**：下层是 OverlayBD 内部的块级 FileCache（7.3 节的 256 KiB 块缓存），管「层的碎片」；上层是镜像缓存服务（src/image/cache/），以 RocksDB 持久化的引用图管「整层」的去留^[22]。两层的策略不同，但哲学一致：**宁可留垃圾，不可删活物**。

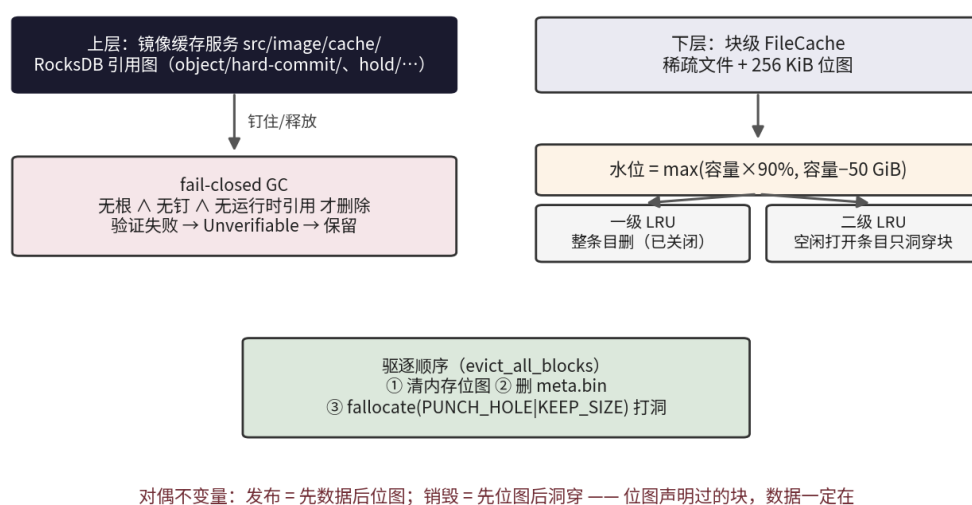


图 7-2 有界缓存的双层结构：块级 FileCache 按 LRU 驱逐，镜像层由 RocksDB 引用图 GC 把关

先看块级的水位公式。FileCache 的高水位不是固定百分比，而是（cache_pool.rs L435-439）：

```
// storage/overlaybd/src/backend/cache/full_file_cache/cache_pool.rs:435-439
pub(crate) fn calc_water_mark(capacity_bytes: u64) -> u64 {
    let ratio_mark = capacity_bytes.saturating_mul(WATERMARK_RATIO) / 100; // 容量 × 90%
    let free_space_mark = capacity_bytes.saturating_sub(MAX_FREE_SPACE_BYTES); // 容量 - 50 GiB
    ratio_mark.max(free_space_mark) // 取两者中更保守（更高）的那个
}
```

即 **水位 = max(容量 × 90%, 容量 - 50 GiB)**。为什么不只用 90%？对小容量盘，90% 只剩几百 MB 余量，碎片与突发写入很快吃掉它；「容量 - 50 GiB」保证无论盘多大都预留绝对余量。另有独立磁盘压力线：可用空间跌破 128 MiB 时无条件驱逐。缓存容量默认 4 GiB、逐出周期 1 秒（cache/mod.rs 常量区）^[1]。

驱逐对象的选择是两级 LRU（最近最少使用）。第一级：已完全关闭（open_count == 0 且无进行中回填）的条目，整条目连目录删除（evict_entry，L623 起）。第二级：仍被打开

但空闲超时的条目——只洞穿其已缓存的块，条目保留、打开中的文件描述符继续可用，下次读自然回源重填（`evict_open_entry_blocks`，L590 起）。**为什么需要第二级？**沙箱 `rootfs` 在整个生命周期内保持打开，只有第一级的话，运行中的沙箱会钉住全部热数据，缓存在高压下毫无还手之力。

驱逐的物理动作是 `fallocate(PUNCH_HOLE | KEEP_SIZE)`（`sys/release_space.rs` L74–79）：把稀疏文件的指定区间打洞，磁盘块与页缓存同时释放，但文件长度不变、`mmap` 不失效，避免 `munmap/remap` 的昂贵与竞态。`evict_all_blocks` 的顺序是精心设计过的（`cache_entry.rs` L498–525）：持写屏障挡住新回填 → **先清内存位图**（从此读未命中、回退源站）→ **再删磁盘 `meta.bin`**（防崩溃后重载到「声称有块、实际已打洞」的位图）→ 最后才 `PUNCH_HOLE`。

上层：RocksDB 引用图与 fail-closed GC。镜像缓存服务把「哪个镜像配置引用哪些层、哪些层被哪些运行时钉住」建模为持久化引用图，落在 RocksDB 封装的 `LocalKvStore` 里（`local_store.rs` L5；`graph.rs` L79，键前缀 `object/hard-commit/`、`hold/`、`ref/config-to-hard/` 等）；编排器在启动、暂停、删除时 `pin/unpin` 更新引用^[1]。GC 删除一层前必须证明它「无根、无钉、无运行时引用」；任何一步验证失败——文件丢失、大小漂移、路径越出提交库、`stat` 失败——都归入 **Unverifiable** 并保留该层（`gc.rs` L33–40 注释明言 `fail-closed`；`service.rs` L622–695）。「fail-closed」是安全工程术语：验证不了就按最保守的方式处理。测试甚至专门断言「无 digest 的本地提交文件必须 fail closed」（`service.rs` L2327、L2354）。**为什么不用简单 LRU 删最旧镜像？**「旧」不等于「没人在用」：一个 `paused` 一个月的沙箱，其层依然活着，删掉它 `resume` 就永久失败。钉是唯一正确的活度判据，LRU 只是无引用时的平局裁决。

【零基础知识补给站】

LRU (Least Recently Used, 最近最少使用)：最常见的缓存驱逐策略——最久没被访问的最先被淘汰。思想类似「书柜放不下时，先扔最久没翻的书」。

RocksDB：Facebook 开源的高性能嵌入式键值数据库，基于 LSM 树，常用于做本地持久化索引^[22]。

引用计数 / 钉住 (Pin)：使用者显式登记「我在用这个资源」，GC 只删没有任何登记的资源。

fallocate / PUNCH_HOLE：Linux 文件系统调用，可以在文件中间「打洞」——洞里的磁盘块被回收，但文件大小不变，读洞会返回全零。稀疏缓存文件就靠它瘦身。

Insight#1 双层缓存的分工值得抄作业：块级缓存管「热度」（LRU + 水位），引用图管「生死」（pin + fail-closed GC）。热度决定驱逐顺序，生死拥有一票否决。

Insight#2 「先清位图再删 meta 再打洞」与 7.3 节「先写数据再置位」构成完美对偶：发布按「数据先于索引」，销毁按「索引先于数据」——崩溃安全的全部秘密就在这两句话里。

7.5 OCI 镜像解析与转换：regctl、格式识别与 turbo-OCI 的拒绝

OverlayBD 镜像终归要从 OCI 生态来。AgentENV 没有自己实现 registry 协议，而是把 **regctl**（regclient 项目的 OCI registry 命令行工具）当作无状态外壳来调（src/snapshot/image_export/regctl.rs）：同 registry 的层复用走 blob copy，跨 registry 用 blob get | blob put 管道流式转发，上传直接从文件描述符读、字节不进进程内存，认证沿用 Docker config^[1]。**为什么不内嵌一个 registry 客户端库？** shell-out 把认证、代理、重试、各 registry 的怪癖全部外包给一个成熟工具，进程隔离还附带内存安全——这是「简单优先」原则的典型应用。

冷启动入口 ImageResolver::resolve 把一个 OCI 引用解析成 overlaybd 块镜像配置 ResolvedBlockImage（src/image/resolver.rs），其中最关键的一步是**介质类型（media type）**识别。AgentENV 能消费三种格式（src/image/oci_image.rs）：

格式	典型介质类型	处理方式
标准 OCI 镜像	application/vnd.oci.image.layer.v1.tar(+gzip/+zstd)	层是 tar 归档，需转换后才能按需加载
OverlayBD 原生	application/vnd.containerd.overlaybd.image.layer.v1.zfile	已是 LSMT 块格式，直接挂索引按需加载
turbo-OCI（加速版）	application/vnd.containerd.overlaybd.turbo.v1+json	明确拒绝：返回 4xx 用户错误

对 turbo-OCI 的拒绝值得细说。turbo-OCI 是 accelerated-container-image 项目用 obdconv --turboOCI 产出的格式：层仍是 tar，附带 overlaybd 索引与 target-digest 注解。AgentENV 的运行时**没有实现** turbo-OCI 读路径，于是选择在 manifest artifactType 与逐层注解两个层面主动探测并明确报错（oci_image.rs L1168-1294），测试还专门断言「turbo-OCI 必须映射为 4xx」（L2147-2163）——归类为用户选错镜像，而非系统故障。**为什么不能「试着读读看」？** 静默误读格式相近但语义不同的层，比直接拒绝危险一个数量级：报错是即时的，数据损坏是潜伏的。标准 OCI 镜像则要走转换：tar 层被解包重写成 overlaybd LSMT 层（oci_conversion_pipeline_benchmark.rs 这条 790 行基准覆盖的流水线）^[1]——转换一次、处处按需加载，「兼容 OCI 生态但不继承其分发方式」。

【零基础知识补给站】

- Registry（镜像仓库）**：存放与分发镜像的服务，Docker Hub 是最著名的一个。企业内部常用 ACR、Harbor 等私有仓库。
- Manifest（清单）**：镜像的「目录文件」，列出它由哪些层组成、各层的介质类型与哈希。
- Digest（摘要）**：内容的哈希值（如 sha256:...），内容不变则摘要不变——这是「内容寻址」的基础。
- tar 归档**：一种古老的文件打包格式，要读其中一个文件往往需要顺序扫描，不支持高效的随机访问——这正是标准镜像必须转换的根本原因。

Insight#1 把 registry 交互外包给 regctl、把格式识别做成显式白名单，体现的是同一条工程原则：边界处用成熟工具，语义处绝不猜测。

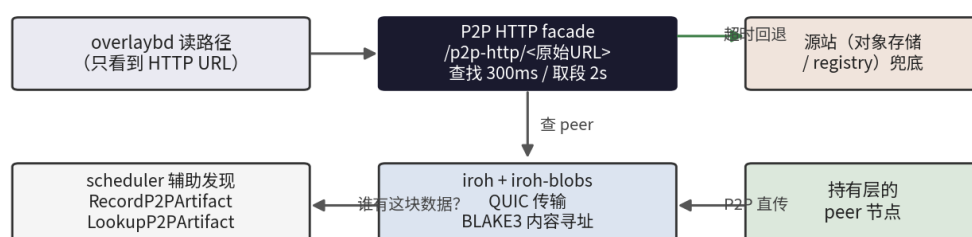
Insight#2 turbo-OCI

的拒绝是一个教科书式的「能力声明」案例：不支持就明确报错并归类为 4xx，而不是含糊地崩溃或静默退化——API 的诚实性和代码的正确性同等重要。

7.6 P2P 分发：iroh-blobs、scheduler 辅助发现与 HTTP facade

按需加载把压力从磁盘转移到了网络，于是新的瓶颈出现：镜像仓库。一个热点模板（比如某个基础镜像刚发布了新快照）可能在一秒被几百个节点回填同一些层，对象存储与 registry 的带宽成为训练吞吐的上限。解法是

P2P（Peer-to-Peer，点对点）分发：已经持有某层的节点直接把它分享给其他节点。



纪律：P2P 只是加速层——超时即回源；描述符 miss 只缓存 5 秒；
传输中途出错绝不回退拼接（防字节流不一致）；发布为 best-effort（半提交窗口存在）

配置 [p2p] 仍为 EXPERIMENTAL，默认关闭

图 7-3 P2P 分发链路：scheduler 辅助发现 → iroh-blobs 直连拉取 → facade 对上层伪装成普通 HTTP

传输层基于 **iroh + iroh-blobs**（src/p2p/iroh/transport.rs L18-25）：iroh 是建立在 QUIC 之上的 P2P 网络栈（自带 NAT 穿透与中继），iroh-blobs 是其内容寻址的块传输协议，以 BLAKE3 哈希寻址 blob^[21]。节点用 FsStore 落盘存储 blob，按 iroh_blobs::ALPN 注册协议处理器，并支持 TempTag 钉住热 blob 防本地 GC。层按 sha256 内容寻址或按 layer UUID 寻址发布（src/snapshot/p2p.rs，key 前缀 snapshot/v1）^[1]。

发现机制借用了集群已有的

scheduler（src/p2p/discovery/scheduler.rs）：节点发布层时通过 gRPC 向 scheduler 登记 RecordP2PArtifact，消费方用 LookupP2PArtifact / ListP2PPeers 查到「谁有这块数据」。**为什么不跑**

DHT（分布式哈希表）自发现？因为集群里已经有一个掌握全量节点信息的 scheduler，复用它既省去一套 gossip 协议的运维成本，又让「数据在哪」与「沙箱调度到哪」的决策可以互相利用。

对上层（overlaybd 读路径）而言，P2P 被**伪装成一个普通 HTTP 源站**——这就是 facade（门面）层（src/overlaybd/p2p/facade.rs）：层 URL 被改写指向本地 facade，路径形如 /p2p-http/<原始URL>；facade 收到 Range 请求后用 reconstruct_origin_url 剥前缀、拼回查询串还原原始 URL（L672-690），先查 P2P，拿不到再回退源站。超时预算很激进：**查找 300 ms、取段 2 s**——P2P 是优化不是依赖，超时即回退，绝不拖慢 guest 的读；描述符缓存 TTL 5 分钟、未命中缓存仅 5 秒（「lookup 超时不缓存 miss，下次读重试」，L358-365），防短暂故障被长时间记住。回退策略还有一条刻意的「不近人情」：**P2P 流一旦开始传输，中途出错绝不回退源站**（测试 p2p_mid_stream_error_does_not_fallback_to_origin, L1144）——Range 语义下无法保证两个源字节流一致，拼接可能产出损坏的层。

这个子系统也有本章最后一个脆弱点：**半提交**。publish_p2p_artifacts 在快照 commit 成功后才以 best-effort 方式向 P2P 网络广告，失败不回滚^[1]；[p2p] 配置至今 EXPERIMENTAL、默认关闭^[1]。「快照已提交但 P2P 未广告」的窗口真实存在，期间其他节点只能回源——功能正确但效率受损。好在 P2P 自始至终只是加速层、正确性由源站兜底，这是它能以「半提交」换取实现简单的前提。

【零基础知识补给站】

QUIC：基于 UDP 的新一代传输协议，内建加密与多路复用，连接建立比 TCP+TLS 快，适合数据中心内高频短连接。

BLAKE3：一种现代密码学哈希算法，比 SHA-256 快得多且可并行，适合给海量数据块做内容寻址。

NAT 穿透 / 中继 (Relay)：让两台都在防火墙后的机器直连或经中转通信的技术。

facade（门面模式）：给复杂子系统包一层简单接口。这里把「P2P 查找 + 回退源站」的复杂逻辑包成「一个普通的 HTTP URL」，overlaybd 读路径完全不知道 P2P 的存在。

best-effort（尽力而为）：做了更好、失败也不影响正确性的操作。

Insight#1 P2P 的设计纪律是「加速层永不承担正确性」：超时 300ms/2s、未命中短缓存、中途失败不回退——每条都在说「宁可回源，不可出错」。

Insight#2 facade 模式让 P2P 成为纯增量：overlaybd 看到的永远是 URL，P2P 可以随时整体摘除（事实上默认就是关的），这是实验性特性该有的接入方式。

本章小结：OverlayBD 用 LSMT

把镜像变成可远程寻址的层栈；按需加载让启动只需索引，块位图 + 选举回填 + 稀疏缓存承接运行时读；双层有界缓存保证磁盘有界且永不错删；regctl 与格式识别接通 OCI 生态；P2P 以 facade + scheduler 发现做横向加速。

第八章 ublk：用户态块设备的高性能 I/O

第七章我们看到 overlaybd 把镜像变成了「可以按需拉取的分层文件」。但还有一个关键环节：Firecracker 的 virtio-block 后端需要的是一个**块设备**（block device，像 /dev/sda 那样的东西），而不是一堆用户态文件。谁把 overlaybd 的用户态逻辑伪装成一块内核认可的磁盘？答案是 ublk——Linux 的「用户态块设备」框架。本章讲清四个问题：为什么偏偏是 ublk；AgentENV 为什么不用现成的 libublk 而是自研了一套框架；为什么 ublk 服务必须是一个独立守护进程；以及这条链路如何与 Firecracker 严丝合缝地对接。

8.1 为什么不用 nbd / tcmu / loop：排除法

「让用户态程序提供块设备」在 Linux 里有好几个老方案。AgentENV 的选择过程本质上是一次排除法，我们逐个过一遍。

方案	工作机制	致命短板
loop	把一个文件回环成块设备	数据必须已是本地完整文件，无法做按需拉取、分层、远程懒加载
nbd	网络块设备，内核通过 TCP/Unix socket 把 I/O 转给用户态服务	协议设计于上世纪九十年代，无多队列、无零拷贝，性能天花板低
tcmu	把 SCSI 命令透传到用户态（Ceph/iSCSI 生态）	走的是 SCSI 协议栈，层数深、开销大，为存储阵列而非本地高速场景设计
ublk	内核 ublk 驱动经 io_uring 与用户态直接对话	需要内核 6.0+，生态年轻——但性能与多队列模型最优

ublk 的核心思想是：内核里有一个极简的 ublk 驱动，它不处理 I/O 逻辑，只负责把块设备收到的每个读写请求「转述」给用户态进程，再把用户态返回的结果交还内核。而「转述」用的通道不是 socket、不是 ioctl，而是 **io_uring**——Linux 新一代异步 I/O 接口，用户态与内核通过共享内存环形队列批量交换请求，一次系统调用可以提交、收割一大批 I/O^{[8][9]}。

对 AgentENV 而言，ublk 还有三个不可替代的特性。其一，**多硬件队列**：一个 ublk 设备可以有多个队列，每个队列绑定独立线程，天然贴合 virtio-block 的多队列并发模型。其二，**零拷贝 / 自动缓冲区注册**（UBLK_F_AUTO_BUF_REG）：数据可以在内核与目标缓冲区之间直接移动，省去传统路径里的内存拷贝。其三，**用户态全权控制**：I/O 到了用户态，就意味着 overlaybd 的分层查找、本地缓存、远程懒加载、限速、录制首触轨迹——一切逻辑都能插在数据路径上。这正是「把镜像供应链架在块设备之下」的前提^{[1][9]}。

【零基础知识补给站】

什么是块设备？操作系统眼里「以固定大小块（通常 512 字节或 4K）为单位随机读写」的设备抽象，磁盘、SSD、虚拟盘都是块设备。文件系统建立在块设备之上。「用户态块设备」则指：设备在内核里注册，但真正的数据处理发生在一个普通用户进程里。

【零基础知识补给站】

什么是 io_uring? Linux 5.1 引入的异步 I/O 框架。用户态和内核共享两个环形队列: SQ (提交队列, 用户放请求) 和 CQ (完成队列, 内核放结果)。相比旧的 read/write 或 epoll 模型, 它把「每个 I/O 一次系统调用」变成「一批 I/O 零到一次系统调用」, 并支持注册缓冲区、注册文件等进一步降开销的机制^[8]。

Insight#1 ublk 胜出的原因不是「更新」, 而是它的通道 (io_uring 共享环形队列) 与模型 (多队列、零拷贝) 与微虚拟机 I/O 的并发形态完全同构。

Insight#2 loop/nbd/tcmu 的淘汰逻辑分别是: 不能懒加载、协议太老、栈太深——排除法比罗列优点更能暴露设计动机。

8.2 ublk 原理与 AgentENV 的自研框架

使用 ublk 的标准姿势是引用官方库 libublk。AgentENV 却另起炉灶, 在 storage/ublk/ 下写了自己的框架。源码开头的注释毫不掩饰动机^[1]:

```
/// The original ublkqueue from libublk is not flexible for our use case,
/// we try to build our own.
—— storage/ublk/src/queue.rs:1-2 [1]
```

「不够灵活」具体指什么? 看自研框架的三个关键设计就明白了。

8.2.1 每个队列一个线程、一个独立的 io_uring

内核为 ublk 设备创建字符设备 /dev/ublkcn 与块设备 /dev/ublkbn, 控制面则通过 /dev/ublk-control 下发 ADD_DEV / DEL_DEV / SET_PARAMS 等命令^[9]。设备建好后, AgentENV 为每个硬件队列 (hw queue) spawn 一个专用线程, 线程内建一个 current_thread 的 Tokio runtime, runtime 里创建一个只属于这个队列的 io_uring (queue.rs:178-184, SQE/CQE 各 depth×2), 并把 /dev/ublkcn 注册进这个 ring (queue.rs:201-203)。「为什么不所有队列共享一个大 ring?」源码注释给出了血淋淋的理由^[1]:

```
// NOTE: we create a separate io uring for each queue.
// I decide to not share one io uring among multiple devices.
// When I use shared io uring + register ublk char device, it will stucked when
// DEL_DEV. Since unregister fd only dec the refcount of current rsrc_node in kernel,
// other devices might hold ref on the rsrc_node (even using different slot in sparse
// file table), preventing the ublk char devices been put.
—— storage/ublk/src/dev.rs:283-288 [1]
```

翻译过来: 共享 ring 时, 注销 fd

只递减内核资源节点的引用计数, 别的设备还持有引用, 导致 DEL_DEV 时字符设备永远释放不掉——直接卡死。这种坑只有真踩过才会写进注释; 它是「每队列独立 ring」这条架构约束的出生证明。

8.2.2 on_thread_park: 线程睡觉前的最后一次提交

io_uring 的提交模型有个细节：往 SQ 里放请求只是写了共享内存，需要一次 submit (io_uring_enter) 内核才来看。高频时每次 send 都提交没问题，但异步运行时里，任务把 SQE 塞进队列后可能就让出线程去睡 (park) 了——如果睡着前不 flush，请求就晾在 SQ 里没人处理。自研框架的解法异常优雅：

```
let rt = tokio::runtime::Builder::new_current_thread()
    .enable_all()
    .on_thread_park(move || {
        UBLK_QUEUE_URING.with(|uring| {
            if let Some(uring) = uring.get() {
                let _ = uring.borrow().submit()
                    .with_context(|| format!("queue {qid} submit for io uring"))
                    .inspect_err(|err| {
                        tracing::error!(?err, "on_thread_park callback failed")
                    });
            }
        })
    })
    .build()
    — storage/ublk/src/dev.rs:220-236 [1]
```

on_thread_park 是 Tokio runtime 的钩子：每当 worker 线程即将休眠，就回调一次。在这里调用 ring.submit()，保证「线程睡着之前，SQ 一定是空的」。零额外线程、零定时器，利用运行时自身节奏完成了提交聚合。数据面 worker (storage/util/src/io_ring/worker.rs:129-146) 用了同一个套路。

8.2.3 override_sqe!: auto buffer registration 的惊险一跃

ublk 的零拷贝模式 (UBLK_F_AUTO_BUF_REG) 要求：内核自己决定用哪个注册缓冲区收数据，用户态要在 SQE 的 addr 字段里填入一个特殊编码值 (ublk_auto_buf_reg_to_sqe_addr)。问题是：Rust 的 io_uring crate 构建 SQE 后并不暴露直接改 addr 的接口。框架的做法是定义一个与内核 io_uring_sqe 内存布局完全一致的 RawSqe，然后 transmute 强转^[1]：

```
#[macro_export]
macro_rules! override_sqe {
    ($entry:expr, $field:ident, $value:expr) => {
        unsafe {
            let sqe: &mut $crate::queue::RawSqe = std::mem::transmute(entry);
            sqe.$field = value;
        }
    };
    ...
}
    — storage/ublk/src/queue.rs:38-56 [1]
```

RawSqe 上方特意保留注释：「A struct with the same memory layout as io_uring_sys::io_uring_sqe. The definition must match the one in io-uring-sys

crate。」这句话既是文档也是警告：**这个 transmute 的正确性依赖上游 crate 的结构体布局永不变更**。一旦 io-uring crate

调整字段顺序或填充，这里就是未定义行为。它换来的收益是真实的——auto buffer registration 省掉了 I/O 路径上的一次内存拷贝——但代价是把一个外部依赖的内部表示变成了本项目的ABI承诺。这是全框架最脆弱的一行宏。

【零基础知识补给站】

什么是缓冲区注册（buffer registration）？io_uring

允许把一批内存缓冲区提前「登记」给内核，之后 I/O

直接用编号引用它们，内核无需每次重新做地址翻译与权限检查。ublk 的 AUTO_BUF_REG 更进一步：由内核在请求到来时自动挑选缓冲区，实现接近零拷贝的数据投递^[9]。

Insight#1 「每队列一个独立 ring」不是性能洁癖，而是踩过 DEL_DEV 卡死坑后的架构红线——读源码注释里的失败史，比读设计文档更能理解系统。

Insight#2 on_thread_park 提交法展示了异步运行时与 io_uring 融合的最小代价方案：不新增线程，借运行时的休眠节奏做 flush。

Insight#3 override_sqe! 是性能与脆弱性并存的典型：unsafe 换来的零拷贝收益真实，但把上游结构体布局当成了隐式契约。

8.3 ublk-daemon：为什么必须是独立进程

AgentENV 的 ublk 服务不跑在主服务进程里，而是一个独立二进制

ublk-daemon，主服务通过 Unix socket 上的 JSON

协议与它通信（storage/ublk-daemon/src/protocol.rs，消息带长度前缀，上限 16 MiB 防脏数据）。主服务启动时经

UblkDeviceManager::init_global_from_config_with_p2p_publish_url

建立全局客户端，客户端甚至负责在 socket 不可用时拉起 daemon

子进程并等待其就绪（client.rs:244 的 wait_for_socket_available）^[1]。

「为什么不把 ublk target 直接跑在主服务的线程里？」排除法给出三条硬理由：

- **设备生命周期长于编排操作**：一个 ublk 设备从沙箱创建活到销毁，跨越无数次 pause/resume/fork 调用；若嵌在主服务里，主服务重启或升级就意味着所有设备一起死。独立 daemon 让「编排层可以重启，磁盘还在」。
- **故障半径**：ublk 数据路径是 unsafe 密集区（transmute、mmap、裸指针），把它关进独立进程，panic 不会波及编排状态机。
- **共享与复用**：多个沙箱可以共享同一个只读 ublk 设备（第九章的 fork 全靠这一点），设备的引用计数、热池、按镜像路径的锁，都需要一个跨沙箱的权威管理者——daemon 就是这个管理者。它还有独立 metrics 端口 :9103^[1]。

协议设计上值得注意的是 RestackSnapshot 请求：daemon 支持对一个**活着的、正在服务 I/O 的**overlaybd 设备执行 close_seal + restack——把当前可写 upper 层封存成新的只读

lower，同时设备继续对外服务。这意味着快照不需要先停机摘盘，磁盘快照与 VM 暂停可以解耦（第九章会看到完整编排）。协议还定义了 RestackSnapshotTerminalFailure 这种「终态失败」错误类型，明确告诉调用方「现场可能已被部分改变，不要盲目重试」^[1]。

daemon 内部还维护**设备热池**：沙箱释放设备时不一定真删除，可以放回热池（PoolConfig 可配，含 startup_prewarm 预热开关），下次同构请求直接取用，省掉 ADD_DEV、建队列线程、注册 ring 的整套开销。配合共享只读设备表（device.rs:189 的 shared_readonly_devices，以 image_config 路径为 key、Weak 引用计数），同一份内存快照被 N 个 fork 子使用时，块设备层面只存在一份^[1]。

【零基础知识补给站】

什么是守护进程（daemon）？在后台长期运行、脱离终端的服务进程。这里的关键不是「后台」，而是「生命周期独立」：ublk-daemon 的生死与主服务解耦，正如数据库的生死与使用它的应用解耦。

Insight#1 独立 daemon 的本质是把「快变的编排逻辑」与「慢变的设备状态」拆到两个进程，让两者可以独立失败、独立重启、独立演进。

Insight#2 RestackSnapshot 让「给活磁盘做快照」成为协议级能力，这是第九章快照流程中 rootfs 路径无需长时间停机的底层支撑。

8.4 与 Firecracker 对接：一块 /dev/ublkN 就是 VM 的磁盘

ublk-daemon 把 overlaybd 镜像服务成 /dev/ublkN 之后，对接 Firecracker 就剩「挂盘」这一件事：PUT /drives/{drive_id} 把设备路径设为 path_on_host。但源码里三个细节值得停留^[1]。

细节一：IoEngine::Async。Firecracker 的磁盘后端有两种引擎：Sync（同步 read/write）与 Async（io_uring）。AgentENV 统一选择 IoEngine::Async（sandbox.rs:2464、2532），让 Firecracker 侧也走 io_uring 访问 /dev/ublkN。于是整条 I/O 路径是：guest virtio 队列 → Firecracker 的 io_uring → 内核 ublk → daemon 的 io_uring → overlaybd。全链路异步、无中间阻塞点。

细节二：限速器在 boot 前下发。add_drive 的注释明确写着 rate_limiter 是「pre-boot limiter」（instance.rs:313），配置由 build_disk_rate_limiter 从 machine.disk_rate_limit 生成（带宽与 IOPS 各自的稳态值 + 突发桶），在 VM 启动前随盘下发。「为什么必须在 boot 前？」因为限速器一旦 attach 就覆盖该盘的整个生命周期，boot 阶段的大量并发读（系统加载、startup pack 首触）也要被计入配额，否则一个沙箱启动风暴就能挤爆宿主磁盘。cfg.rs 还对配置做一致性校验：设了突发却不设稳态值会直接拒绝启动（cfg.rs:1090 起）^[1]。

细节三：恢复快照后不能加盘，所以槽位要预留。Firecracker 从快照恢复时，VM 的设备拓扑已被 vm_state.bin 冻结——不能往一台「恢复中」的 VM 里新增物理盘。AgentENV 的对策是 volume_drive_slots：创建时声明「我未来最多挂 N 块卷」，VM 启动时就带上 N 个占位盘槽（config.rs:155「reserved launch-time volume

slots」)。fork 时同理：snapshot_config_for_fork 只能**替换** launch-time 卷槽的内容，不能新增物理盘^[1]。「为什么不动态热插？」因为 Firecracker 快照模型不支持，与其在恢复路径上做不兼容改造，不如在创建时多留几个槽——用一点点静态冗余换取快照/分叉语义的一致性。

【零基础知识补给站】

什么是速率限制器（rate limiter）？令牌桶算法：桶里按固定速率补充令牌，每次 I/O 消耗令牌，没令牌就等待。带宽桶限字节/秒，IOPS 桶限操作数/秒，突发桶允许短时超速。多租户宿主机用它防止「吵闹邻居」把磁盘打满。

Insight#1 全链路 io_uring（Firecracker Async 引擎 + ublk 通道 + daemon 数据面）消除了 I/O 路径上的所有同步阻塞点，这是沙箱磁盘性能逼近本地的结构性原因。

Insight#2 「恢复快照不能加盘 → 创建时预留槽位」是一次典型的「用静态冗余换动态一致性」：接受少量浪费，避免在冻结的状态上做危险修改。

8.5 Direct I/O 与双重缓存消除

最后一个性能话题：缓存。一个 guest 读一个块，数据可能被缓存三次——guest 的页缓存、宿主的页缓存（落在 /dev/ublkN 上）、overlaybd 自己的本地缓存。「双重缓存」（double caching）指同一份数据在宿主页缓存和 overlaybd 缓存里各存一份，内存白白翻倍，还制造一致性问题。

AgentENV 的解法是让 overlaybd 的数据文件走 **O_DIRECT**（Direct I/O）：打开文件时带 O_DIRECT 标志，读写绕过宿主页缓存，直接落盘/读盘（storage/overlaybd/src/backend/local.rs:94「Enable O_DIRECT for aligned I/O」、sys/open_flags.rs:47 DIRECT_IO_OPEN_FLAG = Some(libc::O_DIRECT)）。页缓存的唯一职责留给「Firecracker mmap 内存镜像」那条路径去用——那里恰恰是 fork 共享所必需的（第九章）。磁盘数据路径则只信 overlaybd 自己的缓存，每层缓存只有一个主人^[1]。

O_DIRECT

不是没有代价：它要求内存地址、文件偏移、长度全部按块对齐，所以框架专门准备了 aligned_buffer.rs（注释：「required by Linux O_DIRECT I/O, which demands that the memory address, file offset, ...」），并在 io_uring 路径上对非对齐请求走 bounce buffer（先读进对齐缓冲区再拷出，local.rs:423-436）。「为什么不让内核帮忙对齐？」内核不会帮忙——O_DIRECT 的契约就是「你保证对齐，我保证直送」，这是用使用门槛换确定性的经典交易。

【零基础知识补给站】

什么是页缓存（page cache）？内核为加速文件 I/O 在内存里保存的文件数据副本，是 Linux「空闲内存越来越少」的正常原因。O_DIRECT 让指定文件跳过这层缓存。好处是避免重复缓存、行为可预测；代价是 miss 时直接读盘，且对齐要求严格。

Insight#1 缓存设计的核心不是「要不要缓存」，而是「每层数据只能有一个缓存主人」：内存镜像归页缓存（为了 fork 共享），磁盘数据归 overlaybd（为了去重与限速），O_DIRECT 是划清边界的手段。



全链路 Direct I/O，避免 guest 页缓存与宿主机页缓存的双重缓存

图 8-1 ublk 数据路径：guest virtio → Firecracker (loEngine::Async) → /dev/ublkbn → ublk-daemon 每队列独立 io_uring → overlaybd (O_DIRECT)

第九章 快照与 Fork：全报告的皇冠

前面所有章节——microVM 隔离、镜像懒加载、ublk 数据路径——都是为了本章铺路。快照（Snapshot）把一台运行中的虚拟机冻成可持久化的状态；分叉（Fork）从一份状态同时长出 N 个独立沙箱。这两个能力把「环境」从一次性服务变成了可序列化、可分支的状态机，也正是 Agentic RL 最渴求的基础设施。本章拆解快照九步全流程、「把内存伪装成块设备镜像」的构造设计、fork 的并发实现，并用第三方实测数据回答一个尖锐问题：fork 真的要 100 毫秒吗？

9.1 为什么快照/分叉是 Agentic RL 的命脉

RL（Reinforcement Learning，强化学习）训练 Agent 的方式是让模型在环境里反复试错：给一个初始状态，让当前策略跑一段轨迹（rollout），根据结果更新策略，再来一轮。这个循环有三个结构性特征，每一个都精准命中快照/分叉：

- **rollout 同源**：大量轨迹从**同一个初始状态**出发（比如同一道编程题、同一个预装了工具的桌面）。没有 fork 时，每条轨迹都要重建环境——装系统、装依赖、起服务，秒级到分钟级；有 fork 时，从快照分支即可，社区分析估算「fork 比重建便宜约 100 倍（ $\sim 100\times$ ）」^[13]。
- **树搜索**：MCTS 类的搜索算法要在同一节点展开多个动作分支，每个分支需要一个「从此刻状态出发」的独立环境——这正是 fork 的定义。
- **多轨迹采样与弹性**：训练 GPU 忙时把环境 pause 出让资源，闲时 resume 继续；中途 checkpoint 可以跨策略共享。社区把这套范式总结为「环境 = 可快速序列化/反序列化的状态机；快照 = checkpoint；fork = 从 checkpoint 分支并行探索」^[13]。

换言之，没有快照/分叉，RL 的环境成本会随轨迹数线性增长并吃掉大部分算力预算；有了它，环境成本趋近于常数。这就是为什么官方把「快速快照、恢复和分叉」列为 AgentENV 的核心能力^[2]，也是本章被称为「皇冠」的原因。

【零基础知识补给站】

什么是写时复制（CoW, Copy-on-Write）？分叉时父子先共享同一份数据，谁要修改谁才复制那一份被改的页。它让「复制」的成本从「复制全部」降到「只复制改动」。本章所有效率奇迹的底层都是它：内存层共享只读、各子只为自己新写的脏页买单。

Insight#1 快照/分叉不是锦上添花的功能，而是把 RL 环境成本从「随轨迹线性增长」压到「接近常数」的结构性杠杆。

Insight#2 「状态机 + 快照」范式相对「服务」范式的跃迁：环境不再是租来的运行时，而是可以 checkpoint 和 branch 的数据。

9.2 快照九步全流程逐步拆解

入口是 FirecrackerSandbox::pause_to_dir (src/sandbox/firecracker/sandbox.rs:1176) , 真正的流水线在 snapshot_to_dir (sandbox.rs:1204) 。结合仓库源码与第三方实测^{[1][13]}, 完整流程可拆为九步:

第 1-2 步: 先刷卷, 再暂停

如果沙箱挂了可写持久卷, 第一步是经 envd RPC 调 sync_writable_volume_filesystems (sandbox.rs:1181-1187) , 把卷上的文件系统刷盘——**文件系统一致性必须先于 VM**

暂停, 否则快照里封存的是半个写事务。然后 fc_instance.pause() (instance.rs:447) 向 Firecracker 发 PATCH /vm 置 Paused: vCPU 线程停止调度, 世界静止了。

第 3-5 步: vm_state.bin、脏范围、读脏页

暂停后进入内存快照的三连击 (sandbox.rs:1362-1377) ^[1]:

```
let firecracker_pid = self.fc_instance.pid()?;
self.fc_instance.create_state_only_snapshot(vm_state_path).await?;
let dirty_ranges = self.fc_instance.get_dirty_memory_ranges().await?;
convert_dirty_memory_to_overlaybd(firecracker_pid, &dirty_ranges, &mem_overlaybd_dir,
memory_output).await
    — src/sandbox/firecracker/sandbox.rs:1362-1377 [1]
```

第 3 步, PUT /snapshot/create 且 snapshot_type=Diff、**不带 mem_file_path**, 只落 vm_state.bin (设备与 vCPU 状态) ——这正是第六章那个「可选 mem_file_path」补丁的用武之地 (instance.rs:468) 。第 4 步, GET /vm/dirty-memory-ranges 拿到脏区间列表 (instance.rs:480) ——这是第二个补丁。第 5 步, 用 process_vm_readv() 直接从 Firecracker 进程把脏页读出来。

第 6-7 步: 写成 overlaybd 层, 叠加父层

脏区间经 dirty_ranges_to_segment_mappings (overlaybd_snapshot.rs:694) 翻译成 overlaybd 的 SegmentMapping 段映射, 再经 convert_dirty_memory_to_overlaybd (overlaybd_snapshot.rs:775) 以 32 并发 compact_to 写成只读层 mem_overlaybd/overlaybd.commit。随后 build_mem_snapshot_image_config (overlaybd_snapshot.rs:574) 把新层叠加到父快照继承的 lowers 之上, 写出 mem_image.json——内存镜像的「层叠清单」就此更新。

第 8-9 步: rootfs 封层与 ublk 暴露

磁盘侧走另一条路: overlaybd 路径下调

restack_snapshot_overlaybd_rootfs (overlaybd_snapshot.rs:637) , 让 ublk daemon 对活着的设备执行 close_seal + restack——把可写 upper 封为新 lower, 再重写 image config; 非 overlaybd 路径则只能 copy_cow 整盘 (sandbox.rs:1278-1286) 。最后组装 SandboxSnapshotManifest 与 FirecrackerSnapshotConfig, 快照产物经由 ublk 设备对外暴露、可被发布。

九步之中隐藏着一个不对称：**磁盘写本就累积在可写 overlaybd upper 里，快照时 seal 即可，无需重新发现拷贝；而内存脏页必须当场从 Firecracker 进程读出来物化成新层。**第三方实测因此得出结论：快照延迟主要随脏内存增长，重磁盘写反而几乎不涨价^[13]。此外，脏页查询的「非清除」语义（6.4 节）保证了任何一步失败都可安全重试，配合 `recover_capture_failure` 与错误语义分层构成完整的失败闭环^[1]。

层数管理同样被设计过：每次快照新增一层， $\text{预算} = 32 - \text{稳定前缀层数} / 4$ （至少 1），超预算时 `rewrite_lowers_with_runtime_roots` (`overlaybd_snapshot.rs:416`) 把运行时后缀层压实成单个 `mem_compacted.commit`，远离 overlaybd 255 层的硬上限 (`DEFAULT_MAX_OVERLAYBD_SNAPSHOT_LAYERS = 32`)^[1]。

【零基础知识补给站】

什么是缺页中断 (page fault) 与懒加载？CPU

访问一个尚未映射的内存页时触发缺页中断，操作系统这时才把数据填进来。懒加载 (lazy loading) 即利用此机制：恢复 VM 时不预读内存，guest 访问到哪页，才现取哪页。AgentENV 恢复路径用 `userfaultfd` (UFFD，把缺页处理交给用户态) 与 File 后端 `mmap` 实现这一点^{[1][10]}。

Insight#1 九步流水线的核心不对称：磁盘「本就分层、快照即封层」，内存「必须当场物化脏页」——这决定了快照延迟的一阶变量是脏内存而非磁盘写。

Insight#2 非清除脏页语义 + 层数预算 +

失败重试闭环，说明这条流水线是按「会失败」设计的，而不是按「会成功」设计的。

9.3 把内存伪装成块设备镜像：全书最优雅的抽象

第 5-7 步里藏着一个值得单独成节的设计：**guest 内存被伪装成了一个 overlaybd 块设备镜像**，从而复用整条为 `rootfs` 打造的供应链。

这个魔法的关键是 `SegmentMapping`——overlaybd 层里「段映射」的编码^[1]：

```
let mut remaining_sectors = length / OVERLAYBD_ALIGNMENT; // 512B 扇区
while remaining_sectors > 0 {
    let sector_count = remaining_sectors.min(Segment::MAX_LENGTH as u64);
    mappings.push(SegmentMapping::new(
        image_offset / OVERLAYBD_ALIGNMENT, // 目的：内存镜像偏移
        sector_count as u32,
        base_host_virt_addr / OVERLAYBD_ALIGNMENT, // 源：Firecracker HVA
        false, 0,
    ));
}
```

—— `src/sandbox/firecracker/overlaybd_snapshot.rs:735-749` [1]

段映射的三元组是「目的偏移、长度、源偏移」。通常源偏移指向层文件内部；而这里源偏移被直接编码成 **Firecracker 进程的宿主虚拟地址 (HVA)**，数据源则是一个实现了 `VirtualFile trait` 的 `ProcessVmReader` (`process_vm_reader.rs:11`) ——它对 overlaybd 的压实器伪装成一个普通文件，内部却用 `process_vm_readv` 读另一个进程的内存（带 `EINTR` 重试、0 字节即报错）：

```
match process_vm_readv(self.pid, &mut local_iovs, &remote_iovs) {
    Ok(read) => read,
    Err(nix::errno::Errno::EINTR) => continue,
    Err(err) => return Err(err).with_context(|| ...),
}
— src/sandbox/firecracker/process_vm_reader.rs:43-56 [1]
```

于是压实器浑然不觉地「读文件」，实际是在做跨进程内存拷贝，并把读到的脏页写成标准的 overlaybd 只读层。校验同样严格：page_size 必须 4096、区间 4K 对齐、目的区间不得重叠、不得越界（overlaybd_snapshot.rs:697-733）^[1]。

这一抽象的收益是惊人的：内存层从此与 rootfs 层**共用同一套分层、压实、ublk 暴露、S3 上传、P2P 分发设施**^[1]。恢复时，ublk-daemon 把内存镜像服务成只读块设备，Firecracker 以 File 后端 mmap 它，缺页驱动懒加载——内存供应链与磁盘供应链在代码层面就是同一条。一个抽象，吃掉两条链路。

【零基础知识补给站】

什么是 process_vm_readv？Linux 提供的跨进程内存读取系统调用：给定目标进程 pid 和一组「远程地址 + 长度」，直接把对方内存拷进自己的缓冲区，无需 ptrace 挂起对方。它要求调用者有权限（CAP_SYS_PTRACE 或同属关系），是快照、调试器类工具的利器。

Insight#1 把 HVA 编码进 SegmentMapping、把进程内存伪装成 VirtualFile——两个 trait 级别的伪装，让内存快照零成本继承了整套镜像基础设施。

Insight#2 优雅的不是技巧本身，而是「没有为内存发明第二套存储系统」：复用即正确性、复用即可维护性。

9.4 fork 的实现：一次暂停，N 个孩子

fork 的后端实现是 FirecrackerSandbox::fork（sandbox.rs:466）。它的结构出奇地简单：

```
let snapshot_config = match FirecrackerSandbox::pause(self).await { ... };
FirecrackerSandbox::resume(self).await.map_err(SandboxCaptureError::terminal)?;
let start_results = futures::future::join_all(children.into_iter().map(|child| {
    async move {
        let mut child = child?;
        match child.start().await { Err(err) => { let _ = child.stop().await; ... } }
    }
})).await;
— src/sandbox/firecracker/sandbox.rs:470-520 [1]
```

一次 pause 产出一份共享快照，随即 resume 源 VM，然后 join_all 并发拉起全部子沙箱。单个子失败只 stop 自己，不株连兄弟。源 VM 的停顿时间被压缩到「一次快照」的宽度，N 个孩子分摊这一次代价^[1]。

成本能做到多低？关键在内存路径：多个子 VM 经 get_or_create_shared_mem（sandbox.rs:2238）**共享同一个只读 ublk 内存设备**，Firecracker 以 File 后端 mmap

它，缺页时从页缓存懒加载——而页缓存是全机共享的。于是 N 个 fork 子的内存加载成本接近

$O(1)$ ：第一个子 miss 时取一页，其余 99 个直接命中缓存。这正是 8.5

节「内存镜像归页缓存管」的回报^[1]。此外 fork 子走 start_resume，可以吃 Firecracker 预热池（sandbox.rs:1993–2017），跳过进程 spawn 与 API socket 轮询。

孩子们共享了内存，却必须拥有独立身份。代码里的「子身份三件套」^[1]：

- **网络 slot**：每个子经 NetworkManager::global().allocate_any() 拿独立 slot（netns + tap0 + IP），load_snapshot_file 用 network_overrides=[("eth0","tap0")] 把恢复 VM 的网卡重定向到新 netns（sandbox.rs:2238–2294）。
- **MMDS sandbox_id**：覆写 mmds_metadata.sandbox_id，guest 内的元数据服务看到的是新身份（sandbox.rs:764–773）。
- **envd token**：编排层为每个子生成独立 envd_access_token（src/orchestrator/service.rs:735），控制面鉴权互不通串。

数量上，API 层硬性限制 $\text{count} \in [1, 100]$ （src/api/impls/sandbox.rs 的 sandboxes_sandbox_id_fork_post, L1439–1443）。注意文档口径曾写「可 fork 多达 16 个子」，源码上限实为 100——以源码为准^{[1][13]}。

【零基础知识补给站】

什么是预热池（warm pool）？提前 spawn 好、待命即取的资源池。AgentENV 的 FirecrackerPool 热条目是 WarmFirecracker { slot, fc_instance, work_dir } 三元组：网络槽位 + 已就绪进程 + 工作目录一体。池子带低/高双水位和「只升不降」的棘轮：观测到突发需求后永久保留更多热容量（warm-pool/src/lib.rs:170–183）^[1]。

Insight#1 fork 的经济学：一次 pause 被 N 个孩子摊销 + 共享只读内存设备让页缓存命中 $O(1)$ + 预热池免 spawn——三层摊销叠出「分叉近乎免费」。

Insight#2 身份与状态的分离是 fork

成立的前提：状态（内存、磁盘）共享，身份（网络、sandbox_id、token）每子独立。

9.5 第三方实测：GenSec 的 fork 延迟实验

官方宣传「增量快照 <100ms」^[2]。GenSec.ai 的实测文章《Inside Kimi K3's AgentENV: Can It Really Fork in 100 ms?》（2026-08-03）提出了一个更诚实的问题：fork 的**完整外部延迟**是多少？实验在同一宿主上进行，每个数据点取 5 个新源的中位数^[13]。

guest 内存	脏内存量	fork 完整延迟（中位）
1 GiB	0 脏页	360 ms
1 GiB	512 MiB	642 ms
4 GiB	512 MiB	715 ms
4 GiB	2 GiB	1.75 s

拟合结果异常干净：**延迟 $\approx 381.6 \text{ ms} + 0.6728 \text{ ms} \times \text{脏内存 MiB}$ ($R^2=0.996$)**。截距约 381.6 ms 是固定开销（pause、vm_state、层写出、子创建编排），斜率约 0.6728 ms/MiB 是脏内存物化的边际成本。另有约 80 ms 发生在 fork API 返回与首次可用之间——大部分增量成本发生在 fork 端点返回之前，即「恢复是懒的，快照捕获不是」^[13]。

这与官方 <100ms

矛盾吗？不矛盾。官方数字是「增量快照操作」的内部边界，实测是完整外部操作（pause + 捕获 + 发布 + resume + 子创建 +

首次可用）。两个数字度量的是不同东西，但实测定量地回答了官方没说的问题：**脏内存是 fork 延迟的一阶变量**，而磁盘写几乎不是——因为磁盘写本就累积在可写 upper 里，快照时 seal 即可^[13]。

与 TClone（基于 CRIU 的容器进程树 fork 方案）的对比更富启发性^{[13][28][29]}：TClone 在 2 GiB 脏内存时 897 ms，斜率约 0.177 ms/MiB，是 AgentENV 的约 1/4——因为它让子直接 CoW 共享源的驻留页，不必当场物化。但 TClone 随**进程数**增长变慢（CRIU 要逐个重建每个进程的资源），AgentENV 对进程数几乎免疫——进程状态编码在 VM 内存里，内核之下一概不管。一句话：AgentENV 用「同步脏内存发布」换「低进程数成本」，TClone 相反。两种答案都不免费，优化的是不同的契约^[13]。

这也是 GenSec 给出的更大教训：任何 fork

系统都有一个不可约问题——「哪些变更状态必须在子可运行前被独立拥有？」AgentENV 答「脏 guest 内存必须先先进入只读层」；TClone

答「子可先共享活页，持久化随后」。选哪个答案，取决于你更怕脏内存还是更怕进程数^[13]。

【零基础知识补给站】

什么是 CRIU？Checkpoint/Restore In Userspace，把 Linux 进程冻成文件再原地/异地复活的工具集。它要逐个处理进程的文件描述符、内存映射、信号、socket——进程越多越慢，这正是 TClone 随进程数劣化的根源^[28]。

Insight#1 实测与官方不矛盾：一个量的是内部快照边界，一个量的是端到端外部延迟；但 0.6728 ms/MiB 的斜率首次定量标定了脏内存的价格。

Insight#2 fork 系统的设计空间被一个不可约问题定义：「什么状态必须在子运行前独立拥有」。比较方案时先问这个问题，再看数字。

9.6 脆弱面：皇冠上的三道裂纹

诚实的技术解析必须以脆弱面收尾。源码分析指出了三处^[1]。

裂纹一：fork 子的 guest 身份同卵

代码中未见对 guest hostname 或机器种子的显式重置——fork 出的 guest 继承源 VM 的 hostname、/etc/machine-id 和内核熵池状态。同一源 fork 出的上百个实例在 guest 视角完全同卵：以 machine-id/hostname 为唯一性依据的软件会隐蔽冲突；同源快照恢复的 VM 熵状态相同，可能弱化早期随机数质量（影响密钥生成等）。目前隔离完全依赖 guest 外机制（netns/IP、MMDS、token）^[1]。

裂纹二：process_vm_readv 阻塞 Tokio worker

ProcessVmReader 的读取是同步系统调用，而 process_vm_reader.rs:20 留有 TODO 自述未做 offload——大脏页集的快照打包会卡住 Tokio worker 线程，影响同线程上其他异步任务。结合 9.5 节「捕获不懒」的结论，这是高脏内存场景下延迟与抖动的一个内部放大器^[1]。

裂纹三：补丁 Firecracker 的升级风险

整条内存快照链路依赖两个上游没有的补丁（/vm/dirty-memory-ranges、可选 mem_file_path），而仓库 thirdparty/ 只 vendored 了生成的客户端，未见补丁源码随仓库管理。升级上游 Firecracker 而补丁未跟随，pause/fork 全链路存在静默退化风险^[1]。

【注意】

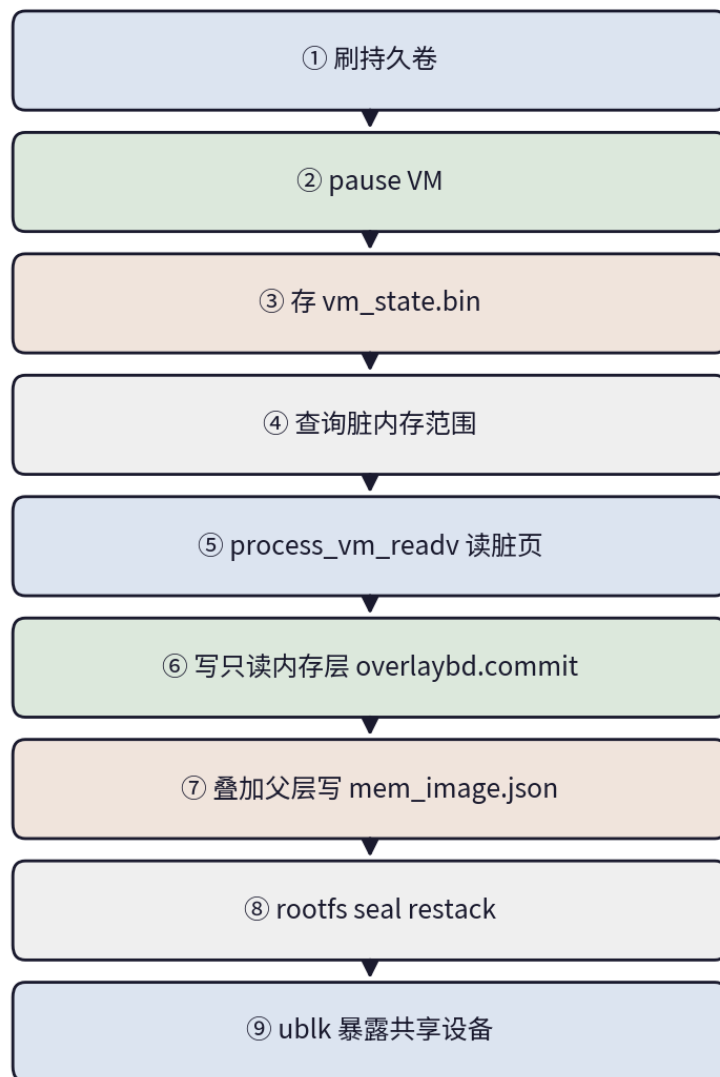
以上三处均为源码审阅结论，未见对应修复；它们的共同点是「平时不出事，出事很隐蔽」。生产部署前应评估：guest 内是否有依赖 machine-id/hostname 唯一性的软件；高脏内存负载下是否观测到控制面延迟抖动；Firecracker 版本升级是否有补丁跟随流程。

【零基础知识补给站】

什么是熵池（entropy pool）？内核收集硬件噪声攒下的随机性储备，/dev/random 等接口从中取数。虚拟机 fork 后，多个子继承了同一份熵状态，早期生成的随机数可能雷同——对密码学用途是隐患。现代内核有若干缓解机制，但快照场景仍需留意。

Insight#1 三道裂纹的共同形态：正确性在 guest 之外（隔离面）坚固，在 guest 之内（身份、熵）依赖约定；性能在主路径优秀，在边界（大脏页集、跨版本升级）有已知缺口。

Insight#2 「皇冠」与「裂纹」并存并不矛盾：快照/fork 的架构优雅是真实的，脆弱面也是真实的——成熟使用者的任务是知道裂纹在哪，而不是假装没有。



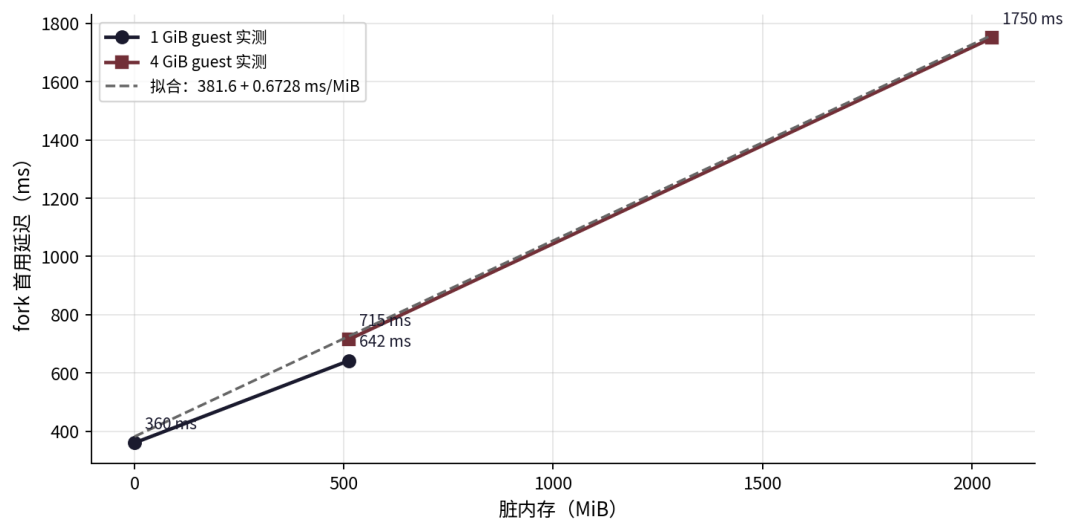
快照只保存脏内存增量，内存层本身也是 OverlayBD 只读层

图 9-1 快照九步流程：刷卷 → 暂停 → vm_state.bin → 脏范围 → process_vm_readv → overlaybd.commit → mem_image.json → rootfs restack → ublk 暴露



一次快照服务 N 个分支：读共享、写时复制，并发启动互不重拷内存

图 9-2 fork 架构：一次 pause 产出共享快照，N 个子共享只读内存设备与页缓存，各自持有独立网络 slot / sandbox_id / token



GenSec 实测口径为完整外部边界；官方 <100ms 指内部快照边界，口径不同

图 9-3 GenSec 实测：fork 延迟随脏内存线性增长，拟合 $381.6 \text{ ms} + 0.6728 \text{ ms/MiB}$ ($R^2=0.996$) [13]

第十章 资源效率与成本经济学

如果说第七章回答的是「环境从哪来」（存储），第九章回答的是「环境怎么复制」（快照与 fork），本章回答的是商业化语境下最尖锐的问题：**这套系统到底省多少钱，省在哪？**官方博客给出的结论是「在典型负载下，AgentENV 将 Agent 执行环境的成本降低了 88.6%–96.8%」^[2]。本章逐层拆解这个数字背后的机制：稀疏性事实（10.1）、pause/resume 经济学（10.2）、内存回收三件套（10.3）、预热池（10.4），然后逐字核对官方的生产数字与定价对比（10.5），最后讨论它隐含的计费哲学（10.6）。

10.1 稀疏性事实：环境大部分时间在等模型

Agentic RL 负载有一个反直觉的特征：**环境是极度稀疏的**。一次 rollout（轨迹采样）中，环境执行一条命令可能只要几十毫秒，然后就要等模型推理出下一个动作——在 GPU 集群上这可能是几百毫秒到几秒。官方博客的表述是：「大多数环境在等待模型生成下一个动作时是空闲的；如果每个环境永久独占自己的 CPU、内存和存储资源，环境成本很快就会成为进一步扩展的瓶颈」^[2]。

把这个事实量化，就是官方定义的**分配-使用比**（allocation-to-usage ratio）：「Resource allocation-to-usage ratio = total resources allocated to all execution environments ÷ actual resource usage on the host」——分配给所有执行环境的资源总量 ÷ 宿主机上的实际资源用量^[2]。比值为 10× 意味着你名义上「挂着」10 倍于物理资源的逻辑环境。传统云厂商按分配计费，稀疏性全部变成浪费；AgentENV 的全部资源效率机制，本质上都是**把稀疏性变现**的手段。

为什么不用「环境数上限」这种粗暴限流？因为稀疏不是均匀的：同一时刻有的环境在编译、有的在等待、有的被暂停。限流牺牲了并发度；而 pause/resume + 内存回收让系统按「实际正在干活」的动态集合分配资源。**为什么不靠超卖（oversubscription）硬撑？**没有回收机制的超卖只是赌博——一旦大批环境同时醒来，OOM（内存耗尽）会随机杀人。稀疏性必须配合「闲置即释放、唤醒即恢复」的快循环才有意义。

【零基础知识补给站】

Rollout（轨迹采样）：强化学习里「让 agent 在环境里完整跑一遍任务」的过程，产生的状态-动作序列用于更新模型。一次训练通常要并行跑成千上万条 rollout。

分配 vs 使用：你向系统「要」了 4 vCPU / 8 GB（分配），但大部分时刻实际只「用」了 0.1 vCPU / 1 GB（使用）。云账单按分配算，物理成本按使用算，差额就是浪费。

OOM（Out Of Memory）：内存耗尽时 Linux 内核会强制杀掉进程腾地方，选谁由启发式打分决定——对训练任务来说是灾难。

Insight#1 稀疏性是 Agentic RL

基础设施的「第一性事实」：所有资源效率设计都是它的推论。看不懂负载，就看不懂架构。

Insight#2 分配-使用比是一个诚实得罕见的指标——它不吹「并发数」，而是直接度量「你为闲置付了多少钱」。

10.2 pause/resume 经济学：<50 ms 恢复与 <100 ms 暂停

稀疏性变现的第一根杠杆是生命周期弹性。官方的口径是：「AgentENV 可以在几十毫秒内启动或恢复基于快照的环境，并在几百毫秒内暂停它们」^[2]；更精确的数字是「基于模板的启动或恢复延迟可低至 49 ms，快照创建延迟可低至 133

ms」^[2]。社区深度实测（GenSec）进一步标定了边界：官方的 <100 ms

是「增量快照操作」的内部口径，完整的外部 fork 延迟随脏内存线性增长（拟合 $\approx 381.6 \text{ ms} + 0.6728 \text{ ms} \times \text{脏 MiB}$ ）^[13]。两个口径不矛盾，但读数字时必须分清。

这套延迟数字为什么能成立？因为 resume

路径上几乎所有昂贵的东西都被预支或共享了：内存恢复走共享只读 ublk 设备 +

UFFD（userfaultfd，用户态缺页处理）懒加载（instance.rs

L491, load_snapshot_uffd），缺哪页才填哪页^[1]；Firecracker

进程、网络槽位、工作目录可以整个从预热池里拿现成的（10.4 节）；rootfs

层在本地缓存里大概率是热的（第七章）。而 pause 释放 CPU 是**自然发生**的——vCPU

线程停止调度后，CPU 时间自动归还，无需任何显式 cgroup 操作^[1]。

经济学含义要算一笔账：如果环境等待模型的时间是 1 秒、恢复只要 50

ms，那么「等待时暂停、动作到达时恢复」的循环开销只有 5%。稀疏性 × 廉价的状态切换 = 逻辑环境数可以与物理资源解耦。官方所说的「让训练系统以更低成本维持数千个逻辑活跃的、有状态的执行环境，而不必为所有环境持续预留全部物理资源」^[2]，落点就是这里。

为什么不直接杀环境、用时重建？重建要重新跑安装依赖、启动服务、恢复上下文，秒级到分钟级，且丢失内存状态——RL 的轨迹连续性被破坏。**为什么不用 cgroup**

冻结（freezer）？freezer 冻住进程但内存照样驻留，只省 CPU 不省内存；pause

同时把脏内存封成 overlaybd 层（第九章），内存也能回收。

【零基础知识补给站】

UFFD（userfaultfd）：Linux 机制，允许用户态进程接管「缺页」处理——访问到还没加载的内存页时，内核不报错，而是通知你的程序去把那一页填上。快照懒加载就靠它。

cgroup freezer：Linux

控制组的冻结功能，让一组进程暂停调度，类似把进程「冷藏」，但内存不释放。

摊销（Amortize）：把一次性的大开销分摊到多次操作上。预热池就是摊销的典型：spawn 进程的成本在平时慢慢付掉，关键时刻免费取用。

Insight#1 pause/resume 的商业价值 = 稀疏负载的套利工具：它把「等待模型推理」这段死时间从成本中心变成了几乎免费的资源出让。

Insight#2 读官方延迟数字务必区分口径：内部快照操作 <100 ms 与外部端到端 fork（含脏内存物化）是两件事，后者的一阶变量是脏内存量^[13]。

10.3 内存回收三件套：balloon 还页、DAMON Reclaim、页缓存复用

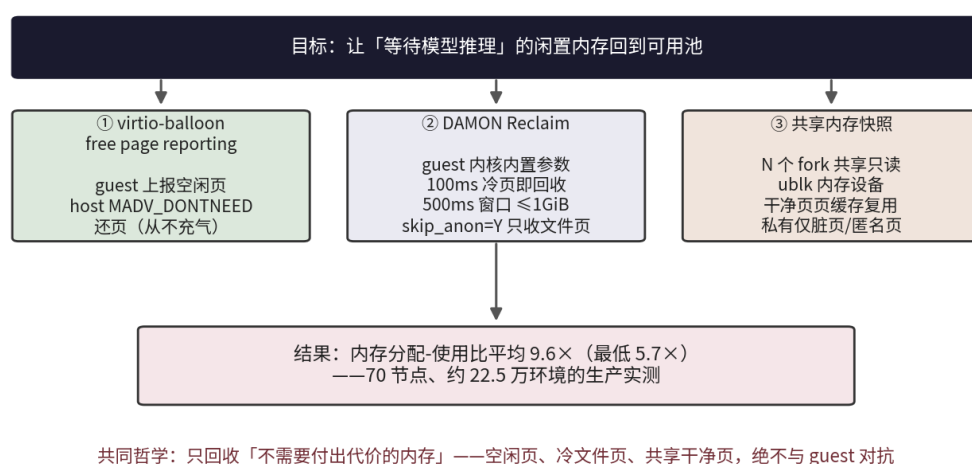


图 10-1 内存回收三件套：guest 内 balloon/DAMON 持续还页，host 侧共享快照页缓存复用

CPU 释放是免费的（线程不调度即释放），内存才是硬骨头。AgentENV 的内存回收是**三件套**，分别从 guest 内、guest 外、跨沙箱三个方向夹击。

第一件：virtio-balloon 的 free page reporting（空闲页上报）。注意 AgentENV 的 balloon 配置（instance.rs L423–435）：amount_mib=0, deflate_on_oom=true, free_page_reporting=Some(true)——它**从不主动充气**。传统 balloon 的用法是 host 向 guest 的「气球」充气，逼迫 guest 挤出内存；AgentENV 只用它的「空闲页上报」附属功能：guest 内核把已经空闲的页主动上报给 host，host 侧对应调用 MADV_DONTNEED 把页还给内核^[1]。**为什么不充气？**充气是与 guest 抢内存的对抗式设计，会引发 guest 内 OOM 与抖动；free page reporting 只捡「guest 自己都不要的页」，零对抗、零惊吓。

第二件：DAMON Reclaim。DAMON（Data Access MONitor，数据访问监视器）是 Linux 内核的访问热度统计框架，其 reclaim 子系统可以按「冷页」策略主动回收^[11]。AgentENV 把整套参数直接编进 guest 内核启动参数（src/sandbox/firecracker/config.rs L43–54）^[1]：

参数	取值	含义
冷页判定窗口	100 ms	页面 100 ms 没被访问就算「冷」，即可回收——极其激进
配额窗口 / 上限	每 500 ms 最多回收 1 GiB	限速器：回收再急也不冲击延迟
水位	990‰ / 200‰	内存占用超 99% 启动回收，降到 20% 停止
skip_anon	Y	只回收 page cache（文件页），不碰匿名页（进程堆栈）

为什么只回收 page cache？匿名页被回收要写 swap，而沙箱没有 swap——碰匿名页等于直接换出失败或抖动；page cache 回收后大不了下次从磁盘（也就是 overlaybd 缓存层）重新读，代价可控。这套参数的精神与 balloon 一致：**只捡软柿子**。

第三件：共享内存快照的页缓存复用。从同一模板恢复的环境共享同一份只读内存快照：「干净页通过宿主机的 Page Cache

复用，只有脏页/写时复制页与活跃匿名内存成为私有分配」^[2]。机制在 `get_or_create_shared_mem` (sandbox.rs L2238)：多个沙箱把同一个 `ublk` 只读内存设备以 `File` 后端 `mmap` 进来，内核页缓存天然去重—— N 个 `fork` 出来的子环境，未修改的内存页在物理上只有一份^[1]。这是 `fork` 「内存加载成本接近 $O(1)$ 」的来源，也是内存分配-使用比能做到 $9.6\times$ 的最大功臣。

为什么不用 KSM（内核同页合并）？KSM 要定期扫描全量内存找重复页，CPU 开销大且安全上有侧信道争议；「共享只读 `mmap`」在设计时就声明了共享，零扫描成本、零扫描误判。

【零基础知识补给站】

virtio-balloon（内存气球）：虚拟化里回收 guest 内存的经典装置——host 给气球「充气」，guest 为给气球腾地方而释放内存。AgentENV 只用它的附属功能「空闲页上报」，永不充气。

MADV_DONTNEED：一个 Linux 系统调用，告诉内核「这段内存我不要了」，内核可以立刻回收对应的物理页。

Page Cache（页缓存）：内核给文件数据建的内存缓存，多个进程映射同一文件时天然共享同一份缓存页。

COW（Copy-On-Write，写时复制）：共享的页只在有人要修改时才复制一份私有副本，没写就永远共享。

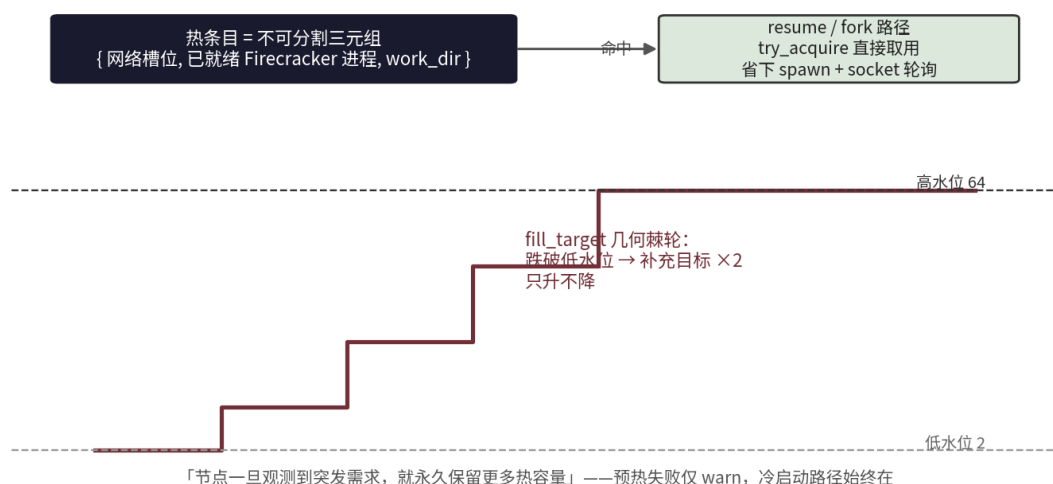
匿名页（Anonymous Page）：不对应任何文件的内存（如堆、栈）；对应的「文件页」则有磁盘文件兜底。

Insight#1 三件套的共同哲学是「只回收不需要付出代价的内存」：空闲页、冷文件页、共享干净页——没有一件涉及与 guest 的对抗，这是它能长期开着而不伤延迟的原因。

Insight#2 内存效率的最大杠杆不在回收而在共享：`fork` 出 N 个子环境但干净页只有一份物理拷贝——设计时共享优于运行时去重（KSM）。

10.4 warm-pool 预热池：几何棘轮与不可分割三元组

`resume` 路径上还有一笔逃不掉的固定成本：`spawn Firecracker` 进程、等它的 `API socket` 就绪、建工作目录、分配网络槽位——加起来几十毫秒，正好是「49 ms 恢复」预算的大头。**warm-pool（预热池）**把这笔钱在平时付掉：后台维护一批「已经 `spawn` 好、`socket` 已就绪」的 `Firecracker` 进程，`resume` 时直接取用（`src/sandbox/firecracker/pool.rs`；通用机制在 `crates/warm-pool/src/lib.rs`）^[1]。

图 10-2 预热池几何棘轮：跌破低水位后补充目标 $\times 2$ 增长，且只升不降

池里的热条目是**不可分割三元组** WarmFirecracker { slot, fc_instance, work_dir }：预建的网络槽位 + 已就绪的 Firecracker 进程 + 工作目录，必须一起取走、一起归还。**为什么不能拆开池化？**因为三者的一致性是在 spawn 时绑定的（进程在对应 netns 内启动、stdio 已重定向进 work_dir），拆开重组等于重新走一遍冷启动。命中条件也因此苛刻：连 capture_output 配置都必须与池一致，否则 stdio 重定向无法重绑。

池的容量策略是全文最有味道的一段代码——**fill_target**

几何棘轮 (crates/warm-pool/src/lib.rs L170–183)：

```
// crates/warm-pool/src/lib.rs:170-183
fn grow_fill_target_after_pressure(&self, pool_len: usize) {
    if pool_len >= self.config.low_watermark || self.config.high_watermark == 0 { return; }
    let mut target = self.fill_target.lock().unwrap();
    let next = (*target).max(1).saturating_mul(2).min(self.config.high_watermark);
    *target = next; // 只升不降：节点一旦观测到突发需求，就永久保留更多热容量
}
```

语义：池子被取到低水位以下（说明刚经历过一波突发），补充目标就翻倍，直到高水位；且**只升不降**。默认低/高水位为 2/64（config/default.toml 的 [pool] 段，network/block/firecracker 三池同机制）^[1]。**为什么只升不降？**突发的持续时间无法预测，降下去再遇到第二波就要重新付冷启动成本；保留热容量的代价只是几个空闲进程与槽位，远小于误判的代价。**为什么是几何 ($\times 2$) 而不是线性 ($+1$)？**突发规模未知时，指数增长用 log 次调整逼近期望水位，线性增长在 N 规模突发下要付 N 次冷启动。这是 TCP 慢启动同款直觉。

两个可靠性细节：预热池是**优化而非必需**——FirecrackerPool::prime 预热失败只 warn 不退出，冷启动路径始终在 (server.rs L103–105) ^[1]；池有自己的单线程 tokio runtime 与 atexit 钩子，保证活得比主 runtime 久、死得比进程干净 (pool.rs L29–42、L59)。

【零基础知识补给站】

预热池 (Warm Pool)：提前创建好一批资源放着备用，类似餐厅高峰期前先煮好一锅饭。

水位 (Watermark)：池子容量的两条警戒线，低于低水位开始补、补到高水位停。

棘轮 (Ratchet)：只能单向转动的齿轮——比喻 fill_target 只升不降的策略。

atexit 钩子：注册在进程退出时自动执行的清理函数，防止热进程变成孤儿。

TCP

慢启动：网络协议中「带宽从低速起步、每轮翻倍」的经典算法，几何棘轮是它的思想亲戚。

Insight#1 预热池把「延迟」从「启动时支付」变成「空闲时预付」——对 RL 这种间歇性突发负载，这是比任何单点优化都划算的买卖。

Insight#2 「只升不降」的棘轮是对「突发不可预测」的诚实承认：与其聪明地预测需求回落，不如笨一点但永不再犯同样的冷启动错误。

10.5 生产数字与成本测算：逐字核对官方口径

本节所有数字**逐字引用官方博客**，属于「特定负载与 2026-07-26 价格下的估算」，请以此前提阅读^[2]。

密度与规模：「在典型配置下，例如一台 128 核 CPU、512 GB

内存的服务器，单节点实测可稳定运行 400 个并发 agent 执行环境，典型环境规格为 2 vCPU / 4 GB 或 4 vCPU / 8 GB。生产环境中，集群已成功扩展到 30,000 个 agent

执行环境，并可随节点增加继续线性扩展。」另有「模板化启动或恢复延迟可低至 49

ms，快照创建延迟可低至 133 ms，并发扇出场景下每个立即可执行子环境的摊销 fork

延迟可低至 122 ms^[2]。注意：128 核带 400 个 2-4 vCPU 的环境，vCPU 超订倍数已在 6-12 倍区间——这只有在 10.1-10.3 的稀疏性机制全部运转时才成立。

分配-使用比：「我们分析了 AgentENV 在高负载时段 70 个节点、约 225,000

个执行环境的真实生产数据。结果显示 CPU 分配-使用比平均 27.9×，最低

14.5×；内存分配-使用比平均 9.6×，最低 5.7×^[2]。社区文章引用的「单机内存超卖

9.6×」正是这里的平均内存比^[16]。CPU 比内存高近 3 倍，与 10.3 的分析自洽：CPU

释放免费且瞬时，内存回收要穿三层机制，天然更保守。

成本测算：「考虑 300 个 agent 执行环境，每个 4 vCPU 和 8 GB 内存，连续运行 720

小时。基于观测到的最低资源分配-使用比，AgentENV 的月度资源成本估计约为

¥15,300。在相同的 CPU

与内存请求假设下，按请求资源计费或依赖长期资源预留的方案，估计成本是 AgentENV 的

8.8 到 31.7 倍。容器实例不参与对比^[2]。「按最低比值估算」意味着这是**保守口径**（用平均比值会更省）。

定价参考表（官方博客 Pricing References, 2026-07-26, 1 USD = 6.773

CNY）逐行解读如下^[2]：

计费模式	CPU 价格	内存价格	计费逻辑解读
AgentENV on ECS	含在 ECS 价格内	含在内	ecs.g9i.32xlarge 整机包月 ¥15,270.53 (128 vCPU/512 GiB)，摊薄到约 ¥0.0707/沙箱小时——成本与跑多少环境无关，只与机器数有关
ACS Container	¥0.06/vCPU · 时	¥0.03/GB · 时	按 Pod limits 之和计费：你「申请」多少就付多少，闲置照付
ACS AgentSandbox	¥0.078/vCPU · 时	¥0.039/GB · 时	按规格计费，但睡眠时免费——已承认稀疏性，但单价上浮约 30%
ECI	¥0.1764/vCPU · 时	¥0.0221/GB · 时	按规格而非实际用量计费，CPU 单价约为 ACS 的 3 倍
AGS Agent Runtime	¥0.2916/vCPU · 时	¥0.09/GB · 时	按分配核数/内存 + 运行秒数计费，是表内最贵的一档
E2B Sandbox	\$0.0504~¥0.3414/vCPU · 时	\$0.0162~¥0.1097/GB · 时	按墙钟运行时间计费——沙箱活着就在烧钱，无论是否在干活 ^[12]
Vercel Sandbox	\$0.128~¥0.8669/vCPU · 时	\$0.0212~¥0.1436/GB · 时	CPU 按实际活跃用量、内存按配置容量计费——半承认稀疏性

把表读成一句话：**所有云产品的计费锚点都是「分配」或「存活时长」，只有自建 AgentENV 的锚点是「实际用量」（通过整机包月 + 内部分配-使用比实现）。**8.8–31.7× 的差距不来自某个单点技术，而来自计费锚点的不同——这正是下一节的主题。

【零基础知识补给站】

vCPU · 时 / GB · 时：云计费的计量单位，1 个 vCPU 跑 1 小时就是 1 vCPU · 时。

ECS / ACS / ECI / AGS：阿里云的产品线缩写，分别是弹性计算（虚拟机）、容器计算服务、弹性容器实例、Agent 运行时服务。

墙钟时间 (Wall-clock Time)：现实世界的流逝时间，区别于「实际干活的时间」。沙箱开了 1 小时但只干了 3 分钟活，按墙钟计费就付 1 小时。

超订 / 超卖 (Oversubscription)：名义分配的资源超过物理总量，赌的是大家不会同时用满。

Insight#1 成本对比的本质是计费锚点之争：按分配计费的产品结构性无法利用稀疏性，哪怕它们的底层技术和 AgentENV 一样先进。

Insight#2 官方特意用「最低分配-使用比」做保守估算、并注明价格日期，是基础设施厂商少有的诚实姿态——但读者仍须记住这是特定负载下的数字，自己的负载要自己测。

10.6 计费哲学：成本应按实际用量而非环境数

把 10.1–10.5 串起来，浮现出一条完整的世界观：**环境是逻辑概念，资源是物理概念，计费应该跟随物理概念。**传统沙箱服务按「环境数 × 存活时长」计费，等于让客户为「等待模型推理」的死时间付费；AgentENV 的答案是让死时间的物理成本趋近于零——pause 释放 CPU、balloon/DAMON 归还内存、共享页缓存去重、预热池摊销启动——于是「维持 22.5 万个逻辑环境」的物理成本只是几十台整机^[2]。

这条哲学对自建 vs 购买的决策有直接影响。买 SaaS 沙箱（E2B 等）时，你的成本曲线是 $O(\text{环境数} \times \text{时长})$ ，稀疏性红利被厂商收走；自建 AgentENV 时，成本曲线是 $O(\text{节点数})$ ，稀疏性红利归你。社区文章 [mushroom.cv](#) 的观察与此呼应：AgentENV 把环境从「服务」模型转为「状态机 + 快照」模型——pause 是「不毁状态地出让资源」，fork 是「从 checkpoint 分支」^[19]。一旦接受状态机模型，「环境数」就失去了成本含义，剩下的只有「同时活跃的状态数」与「状态切换的摊销成本」两个物理量。

但也要诚实地补上另一面：这套哲学成立的前提是**你有足够稀疏且足够大规模的负载**。如果你的环境本来就 100% 忙（例如编译农场），分配-使用比接近 $1\times$ ，AgentENV 的成本优势就只剩技术密度本身；如果你的规模是几十个环境，自建集群的运维成本会吞掉全部红利（详见 12.5 节）。计费哲学的反面不是「错」，而是「有适用域」。

【零基础知识补给站】

SaaS（软件即服务）：直接使用厂商托管的服务并按量付费，如 E2B——你不管机器，厂商管。

自建（Self-hosting）：自己买/租机器部署开源软件，成本是整机价 + 运维人力。

状态机模型：把对象看作「一组状态 + 状态之间的迁移」。环境 = 内存 + 磁盘的状态，pause/fork/resume 都是状态迁移操作。

Insight#1 「按实际用量计费」在云上喊了十年没实现，AgentENV 用架构实现了它——有时计费革命不靠账单系统，靠把闲置成本物理消灭。

Insight#2 稀疏性红利的归属是自建 vs SaaS

决策的隐藏变量：负载越稀疏、规模越大，自建越划算；反之 SaaS 的省心是真金白银买得值的。

本章小结：资源效率的因果链是——稀疏性事实（10.1） $\times$ 廉价状态切换（10.2） $\times$ 内存三件套（10.3） $\times$ 预热池摊销（10.4）= 生产实测的 $27.9\times/9.6\times$ 分配-使用比（10.5）= 计费哲学的重构（10.6）。下一章看数据面与集群层如何把这些单机能力组合成可运营的集群。

第十一章 集群层：Gateway、Scheduler 与 Kubernetes

第五章讲了单机节点的 Rust 控制面，本章把视角拉大到集群：Go 写的 Gateway 与 Scheduler 如何把多台节点粘成一个集群，Kubernetes（下称 K8s，容器编排系统）上的部署形态，以及跨语言的可观测性设计。集群层代码在 `services/` 目录，是一个独立的 Go module^[1]。

11.1 为什么集群层用 Go 而节点用 Rust

AENV 是「集群层 Go + 节点层 Rust」的双语言架构，这不是随意选择。用「为什么不 X」排除法推演设计逻辑：

- 为什么节点不用 Go？节点层要直接操控 KVM、ublk 块设备、netns、Firecracker 进程与 UFFD（userfaultfd，用户态缺页处理）懒加载——这些是高频、精细的系统调用与内存管理，需要精确的内存控制（jemalloc 调优、RAII 资源句柄、零拷贝），且单节点要承载成千上万个沙箱的并发状态机。Rust 的所有权模型与无 GC 停顿更匹配这一层。
- 为什么集群层不用 Rust？Gateway 与 Scheduler 是典型的网络服务：HTTP 反向代理、gRPC 服务、K8s informer watch、Redis 客户端。Go 在这一领域的生态（net/http、grpc-go、client-go）成熟度碾压 Rust，且集群层不触碰底层虚拟化，GC 停顿在代理延迟预算里可以接受。
- 为什么不统一语言省心智成本？两层的质量属性不同：节点层要极致性能与内存安全，集群层要开发效率与生态。统一任何一端都要在另一端付出代价；双语言以「共享 protobuf 契约目录 `services/api/proto/`」为代价换取各自最优，`scheduler.proto` 生成的 `pb.go` 是两语言间的唯一接缝。

【零基础知识补给站】

GC 停顿（Garbage Collection pause）是 Go/Java 这类带垃圾回收语言周期性暂停业务逻辑回收内存的抖动。对代理来说几十微秒无所谓，对要精确控制 guest 内存的虚拟化层则是不可预测的干扰源。

Insight#1 语言边界 =

质量属性边界：性能与内存安全在下沉层（Rust），生态与迭代速度在聚合层（Go）。

Insight#2 共享 proto 目录而非共享代码库，让两个 module 独立演进，契约变更必须显式走 proto 文件——这是多语言系统最低摩擦的协作方式。

11.2 Gateway：三种路由来源与聚合语义

Gateway（`services/gateway/`，核心 `internal/server.go` 约 1008 行）是集群的 HTTP 入口，承担两件事：数据面请求按沙箱 ID

路由到对应节点；控制面请求实时调度或聚合。路由有三种来源，优先级从高到低：

路由来源	形式	说明
Host 头（优先）	{port}-{sandboxID}.{proxy_domain}	域名路由；沙箱 ID 需满足 RFC 952/1123 DNS label 且 ≤63 字符
Header	x-agentenv-sandbox-id / e2b-sandbox-id	双套头兼容 E2B SDK
Path	如 /sandboxes/{id}/pause	控制面路径自带沙箱 ID

表 11-1 Gateway 三种路由来源（services/gateway/internal/server.go）

集群聚合 API（GET /sandboxes、GET /v2/sandboxes）fan-out

到所有节点合并结果，且严格

all-or-nothing——任一节点失败则整体报错。为什么不部分返回？因为 RL 训练控制循环依赖「看到的沙箱列表是完整的」这一假设，部分结果会让上层误判环境数量，宁可显式失败。GET /nodes 系列则经 Scheduler 解析后代理。

两个工程细节值得注意：其一，刻意不用 Go 标准库的 http.ServeMux——ServeMux 会归一化路径中的

%2F（编码的斜杠），破坏代理语义，所以路由是手写的；其二，request_timeout 不会截断 WebSocket 与 streaming 连接（PTY 与日志流是长连接），debug 模式还会回写 x-agentenv-node-id 方便排查路由。

创建请求上 Gateway 还会做「调度提示」：POST /sandboxes-cold 时解析请求体（上限 64 KiB，防止恶意大请求撑爆内存）提取 cpu/memory/images 组成 NewColdSandbox hint 交给 Scheduler；创建成功后再从响应体提取沙箱 ID 回写 RecordAssignment 登记绑定（internal/schedule_hint.go）。认证方面 Gateway 只对控制面校验 X-API-Key（crypto/subtle 常数时间比较）；数据面不认证——私有 ingress 的 e2b-traffic-access-token 与 secure 沙箱的 X-Access-Token 策略只在节点本地执行，Gateway 不重复验。

【零基础知识补给站】

DNS label 限制（≤63 字符、只能字母数字连字符）来自 DNS 协议本身。因为 Host 路由要把沙箱 ID 编进域名，ID 必须是合法 DNS 片段——这就是为什么 host 路由对 sandbox ID 有额外格式要求。

Insight#1 「数据面认证下沉到节点」是责任归位：流量访问策略本来就是节点本地元数据（allow_public_traffic 等），Gateway 重复验只会引入状态同步问题。

Insight#2 64 KiB 调度提示上限是典型的防御性编程：解析用户请求体永远要有界。

11.3 Scheduler：节点发现与两阶段调度

Scheduler（services/scheduler/，核心 internal/service.go）是 gRPC 服务，方法集覆盖调度、节点管理、心跳与 P2P artifact 索引：

gRPC 方法	职责
Schedule	两阶段调度：先过滤再选择
RecordAssignment	登记「沙箱 ID → 节点」绑定
LookupNode / GetNode / ListNodes	按沙箱查节点（数据面路由的依据）
Heartbeat	节点心跳，携带沙箱名单做 Reconcile，响应下发 CPU 模板交集
ReportSandboxEvent / ListObservedNodes	事件上报与观测节点列表
UnregisterNode	节点摘除
ListP2PPeers / RecordP2PArtifact / ForgetP2PArtifact / LookupP2PArtifact	P2P 分发索引

表 11-2 Scheduler gRPC 方法表（services/scheduler/internal/service.go）

节点发现有两种来源：static（配置文件写死节点列表，适合小集群）与 kubernetes（watch EndpointSlice + Pod informer，指数退避重连 1s→30s，支持 IPv4/IPv6）[26]。K8s 模式还有两个 selector 旋钮：ignore_pod_selector 把某些 Pod 完全排除在节点集合外；no_schedule_pod_selector 保留节点但标记为禁调度（比如节点维护期）。

调度本身是两阶段的：

```
eligible := FilterByResourceLimit(rich, s.resourceLimit) // 先过滤
node, selectErr := s.strategy.Select(eligible, req.GetHint()) // 再选
// services/scheduler/internal/service.go, Schedule
```

- 第一阶段资源护栏过滤（internal/filter.go）：max_sandbox_count、max_sandbox_starting_count、max_cpu_used_percent、max_cpu_allocated_percent（可 >100 允许超卖）、max_memory_* 等，另有 including paused 三项把暂停沙箱的资源预留也计入；全部被过滤掉则返回 gRPC Unavailable。
- 第二阶段策略选择（internal/strategy.go）：内置 round_robin（默认，原子计数器）与 random。Strategy 接口接收 RichNode（身份 + 最新心跳快照），内置策略暂不使用快照，但自定义策略可做负载感知调度——接口预留了扩展点。

绑定存储可插拔：默认内存；scheduler.redis_addr 非空即启用 Redis[27]，绑定 TTL 由 binding_ttl 控制，心跳携带的沙箱名单用于 Reconcile 校正漂移。HA 形态是一个 primary（读写）加多个 --query-only 只读副本（仅响应 LookupNode，直连 Redis）——HA 只保数据面路由，primary 宕机时创建沙箱等控制面操作仍不可用。为什么不追求控制面 HA？因为控制面短暂不可用只影响新沙箱创建，而数据面路由中断会让所有存量沙箱失联——两者的爆炸半径差一个量级，这是有意识的取舍。

最有特色的是 CPU 模板交集（internal/cpu_template.go）：各节点心跳上报 Firecracker 的 cpu_config_json，Scheduler 对 CPUID/MSR modifiers、KVM capabilities 做按位 AND 交集，经 HeartbeatResponse.CpuConfigJson 下发。新构建的模板据此只使用集群所有节点都支持的 CPU 特性，保证快照可以跨节点迁移。为什么不取并集？并集会让模板用到某节点没有的指令，迁移即崩溃；AND 交集是「最小公分母」策略，牺牲峰值特性换全域可调度。

【零基础知识补给站】

EndpointSlice 是 K8s 用来记录 Service 后端 Pod IP 的资源对象；informer 是 client-go 的「watch + 本地缓存」机制，比轮询 API server 高效。gRPC 的 Unavailable 状态码语义是「服务暂时不可用，可重试」，比 500 更准确地表达「暂时没有节点能满足」。

Insight#1 「心跳携带沙箱名单做

Reconcile」把绑定存储的最终一致性兜底交给了最频繁的既有通道，零额外组件。

Insight#2 round_robin 用原子计数器实现，无锁、O(1)，与「调度决策要快」的定位一致；复杂负载感知被显式推迟给自定义策略。

11.4 Kubernetes 部署形态

deploy/k8s/ 目录给出了完整的 K8s 部署清单（kustomize base + overlays/default 与 overlays/local-dev 两套 overlay）[26]：

组件	K8s 资源形态	要点
gateway	Deployment + ClusterIP Service	无状态，可水平扩
scheduler	单副本 Deployment + ClusterIP Service	内存绑定时必须单副本
agentenv-node	特权 DaemonSet	挂 /dev/kvm、hostPath /var/lib/agentenv；AENV_NODE_ID 取 Pod 名
agentenv-nodes	headless Service	供 EndpointSlice 节点发现

表 11-3 K8s 部署形态（deploy/k8s/）

节点用特权 DaemonSet（DaemonSet 保证每台物理机恰好跑一个副本的 K8s 控制器）是必然选择：沙箱需要 /dev/kvm 与宿主目录持久化，且一个节点一个 aenv-server 进程管理全部沙箱。headless Service（无 ClusterIP、DNS 直接解析到 Pod IP 的服务）让 EndpointSlice 能枚举所有节点。为什么不给节点也做负载均衡 Service？因为节点不是「可互换的无状态后端」——每个沙箱绑定特定节点，调度必须显式进行，K8s 负载均衡反而会破坏这个语义。

没有 K8s 时可用 Docker Compose 模拟：deploy 目录提供 gateway + scheduler + node-a/node-b 双节点组合，make deploy-up / deploy-down 一键起停，足够本地验证集群行为。

【零基础知识补给站】

DaemonSet 与 Deployment 的区别：Deployment 按副本数调度到任意机器，DaemonSet 强制每台（或每类）机器一个副本。虚拟化节点服务属于后者——它管理的是这台机器上的 KVM 与网卡，搬不走也不能有两个。

Insight#1

scheduler「内存绑定时必须单副本」的注释提醒我们：可插拔存储不只是灵活性，也是 HA 的前置条件——要 HA 就必须配 Redis[27]。

Insight#2 compose

双节点模拟让「集群行为」可以在笔记本上测试，降低了分布式逻辑的调试门槛。

11.5 可观测性：跨语言的指标对齐

可观测性横跨 Go 与 Rust 两个世界，AENV 的做法是用「桶对齐」保证语义一致：Rust 侧 crates/observability 的 Prometheus exporter 定义全局时长直方图 1ms–30min 共 21 档（zfile 两个指标用微秒级细桶），代码注释明确要求与 services/shared/observability/buckets.go 保持同步——跨语言桶对齐让同一组 Grafana 面板能同时展示两侧指标。为什么不各定义各的？Prometheus 直方图的桶边界一旦不同，跨服务聚合的 p99 就失去可比性，「一桶打天下」是最便宜的一致性方案。

节点侧的心跳上报器（src/observability/reporter.rs）走 tonic gRPC 向 Scheduler 上报，超时 10 秒、最大退避 60 秒，上报内容含沙箱 roster 供绑定 Reconcile；src/observability/（1716 行）还包含请求时主机采集（host.rs）与预投影 NodeSnapshot 供 admin API（service.rs）。ublk daemon 在 :9103 独立暴露指标，Go 侧 gateway/scheduler 各有独立 metrics 端口——每个进程一个 /metrics，互不干扰。日志栈也是双轨：Go 用 zap，Rust 用 tracing。

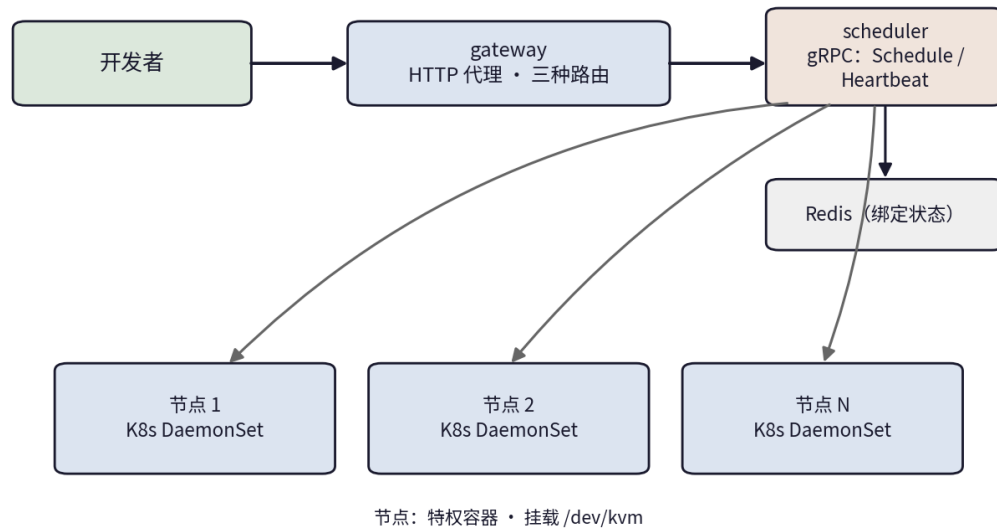
指标体系里有个容易被忽略的细节：Rust 侧 NodeIdentity 的 commit 是编译期注入的（option_env!("AENV_GIT_COMMIT")，src/identity.rs L8–13），并有专门测试确保运行时环境变量无法伪造（L93–116）。为什么如此较真？因为指标上的版本标签是「线上问题定位到具体代码」的唯一桥梁，若可被运行时覆盖，排障时会得到错误的版本结论。

【零基础知识补给站】

Prometheus 直方图（histogram）把观测值按预定义桶（bucket）计数，用 histogram_quantile 估算分位数。桶边界是 schema 的一部分：两侧桶不同，算出来的「p99 延迟」就不是同一个东西。心跳（heartbeat）是节点周期性向调度器报告「我还活着、我有多少资源、我跑哪些沙箱」。

Insight#1 「每桶对齐」的约定靠代码注释维系，跨语言契约没有编译期保证，属于「纪律型一致性」——好在变更频率极低，人工同步成本可接受。

Insight#2 心跳上报器把「绑定校正」搭在既有心跳上，是第十一章与第五章呼应的设计：所有一致性兜底都尽量复用已经在跑的通道，而不是新增组件。



scheduler 采用 primary + query-only 副本实现 HA；gateway 按路由策略转发到节点

图 11-1 集群层架构：Gateway、Scheduler 与节点 DaemonSet（示意）

第十二章 工程实践、安全模型与局限

前十章讲的是「这套系统有多强」，本章讲「它凭什么可信、边界在哪里」。我们分四步：测试体系（12.1）回答「正确性如何被验证」；CI 与工程文化（12.2）回答「质量如何被制度化」；安全模型（12.3）回答「攻击面如何被收敛」；已知局限与适用边界（12.4–12.5）回答「什么时候不该用它」。对零基础读者，这一章其实是全书的「工程方法论」浓缩——测试分层的思路、安全设计的取舍、写局限的诚实，都可以直接搬进你自己的项目。

12.1 测试体系：从 Criterion 基准到变异测试

AgentENV

的测试不是一堆散乱的单元测试，而是一个**分层金字塔**，每层针对不同故障模式^[1]。

第一层：性能回归——四组 Criterion 基准。Criterion 是 Rust

生态的统计学基准测试框架（自动做采样、离群点剔除、回归检测）。仓库里四组 bench 各司其职（crates/benchmarks/benches/）：snapshot_benchmark.rs（667 行：1 GiB 磁盘/1 GiB 内存快照创建、热/冷恢复、并发恢复）、ublk_overlaybd_benchmark.rs（1001 行：块设备与镜像读路径）、orchestrator_store_benchmark.rs（440 行：CAS 迁移、1000 沙箱扫描）、oci_conversion_pipeline_benchmark.rs（790 行：OCI→overlaybd 转换流水线）。**为什么基础设施项目必须先有基准再有优化？**因为第九章那种「49 ms」的承诺，没有持续基准就是一句口号；有了基准，每次提交的性能漂移都无处遁形。

第二层：端到端——MinIO fixture 打穿存储链路。snapshot_oss_e2e_test.rs（794 行）用 crates/test-support 提供的 MinIO（开源的 S3 兼容对象存储）fixture 在 CI 里起真对象存储，端到端验证 OSS 后端快照：ZFile 压缩的 LSMT 层、sha256 校验、P2P 分发、发布/解析全流程^[1]。**为什么用 MinIO 而不是 mock S3？**mock 只能验证「你调了正确的

API」，无法验证「分片上传在断点续传下的字节序正确性」这类真实协议行为。fixture 化的真实依赖是 e2e 测试的性价比甜点。

第三层：特权集成测试。主仓的集成测试需要 root 权限、/dev/kvm 与 netns 操作能力，配套 scripts/run-with-capabilities.sh 做精细提权^[1]。**为什么不全部 mock 掉 KVM？**ublk、netns、Firecracker 的故障大半发生在与内核的边界上（ioctl 语义、能力集、命名空间传播），mock 掉的边界就是测试的盲区。代价是这类测试没法在普通开发机上随手跑——这是显式接受的 trade-off。

第四层：变异测试（mutation testing）与务实的覆盖率。adev 工具链内置 adev mutants（基于 cargo-mutants：自动给源码注入「把 && 改成 ||」「删掉一行」这类变异，看测试套件能不能抓到变异体）与 adev coverage（cargo-llvm-cov）。而覆盖率 CI 刻意设为 **non-blocking**（continue-on-error，注释自承 "informational"，badge 发布到孤儿分支 coverage-data）^[1]。**为什么覆盖率不卡门？**覆盖率度量的是「代码被执行过」，不是「行为被验

证过」；把它做成硬门禁只会催生「为了覆盖而覆盖」的空测试。变异测试反过来度量「测试真的在断言东西吗」，是更有信息量的质量信号——先把贵的信号建好，再把便宜的信号降级为参考，这个排序很见功力。

【零基础知识补给站】

基准测试（Benchmark）：测「多快」而非「对不对」的测试，重点在统计学上稳定可复现。

Fixture（测试夹具）：测试前搭好的环境（比如一个真实跑起来的 MinIO 容器），测完拆掉。

变异测试：故意往代码里「埋 bug」（变异体），看你的测试能不能发现。抓得住变异体的测试才是真测试。

/dev/kvm：Linux 硬件虚拟化接口，没它就跑不了 Firecracker。

continue-on-error：CI 配置项——这个任务失败不算整个流水线失败，只是记录一下。

Insight#1 四层测试各管一类故障：基准管性能漂移、fixture e2e 管协议正确性、特权集成管内核边界、变异测试管「测试本身的有效性」——分层不是堆量，是正交覆盖故障模式。

Insight#2 「覆盖率 non-blocking + 变异测试进工具链」的组合是对度量指标 Goodhart 定律（指标一旦成为目标就不再是好指标）的成熟回应。

12.2 CI 全景与工程文化

仓库 `.github/workflows/` 下共有 **13 个 workflow**
文件（笔者按仓库实际清点；早期分析素材列出 10 个，以仓库为准）^[1]：

Workflow	职责
ci.yml	主门禁：编译、单元测试、lint
coverage.yml	覆盖率采集（non-blocking，badge 发 coverage-data 孤儿分支）
integration-tests.yml	需 root + /dev/kvm 的特权集成测试
ublk-tests.yml	ublk 用户态块设备子系统专项
envd-tests.yml	沙箱内 envd（进程/文件/PTY 控制面）测试
e2b-benchmark.yml / benchmark.yml	性能基准与 E2B 兼容性基准
mutation-tests.yml	cargo-mutants 变异测试
services-ci.yml	Go 集群服务（gateway/scheduler）测试，CI 中起 redis-server
docs.yml	mdBook 文档构建（docs/src 共 66 个 md）
open-code-review.yml	自动化代码评审
publish-tools-image.yml / release.yml	工具镜像发布与版本发布

注意 CI 的结构与架构同构：Rust 节点、Go 服务、文档、性能、安全各有独立流水线，互不阻塞——覆盖率失败不会挡住 mutation-tests

的结果。**为什么不合并成一条大流水线？**因为不同子系统的故障率与运行时长差异巨大（变异测试可能跑几小时），拆分让反馈环最短化：改文档的人不用等 KVM 测试。

比流水线更有信息量的是**工程文化文本**。CLAUDE.md（242 行）是仓库里唯一的 AI 协作规范，四条原则：**Think Before**

Coding（显式声明假设、不确定就停下来问）；**Simplicity First**（原话级别的要求：「200 行能写成 50 行就重写」）；**Surgical Changes**（每行改动可追溯到需求）；**Goal-Driven Execution**（先写失败测试再实现）^[1]。CONTRIBUTING.md 则规定：bug 报告必须带精确 commit、内核/虚拟化环境与最小复现；安全漏洞走私下 advisory 而非公开 issue；大改动须先达成设计共识；提交信息走 Conventional Commits（「feat:/fix:/refactor:」前缀规范），changelog 由 cliff.toml 自动生成；**生成代码只允许改 schema 后用 make 重新生成**，严禁手改（OpenAPI 生成的 server 路由、Firecracker 客户端都是生成物）^[1]。

「生成代码只改 schema」这一条值得单独称重。**为什么不能手改生成物？**手改会在下次重新生成时静默丢失，且评审者无法判断改动是来自 schema 还是手工。把「单一真相源」原则贯彻到代码生成层，是大型 API 兼容层（第三章的 E2B 兼容）能长期维护的前提。

【零基础知识补给站】

CI（持续集成）：每次提交代码就自动跑一遍编译与测试的流水线，GitHub 上常用 GitHub Actions。

Conventional Commits：一种提交信息格式约定（如 feat: 新功能、fix: 修 bug），好处是可以工具自动生成更新日志、自动判断版本号。

孤儿分支（Orphan Branch）：git

里一个没有历史根源的分支，常用于存放与代码无关的产物（如覆盖率 badge 数据）。

Schema：接口的结构化定义文件（如 OpenAPI yaml）。「改 schema 重新生成」= 只维护定义，代码由工具产出。

Insight#1 CI 结构与系统架构同构（康威定律的良性应用）：子系统各自独立流水线，反馈环最短，故障域也最小。

Insight#2 CLAUDE.md 四原则全部指向同一件事——约束「熵增」：声明假设防误解、强制简单防膨胀、手术式改动防漂移、测试先行防返工。这套纪律对人和对 AI 同样有效。

12.3 安全模型：认证、流量与文件系统的三层加固

先给出最重要的警告，再讲机制：

【注意】

AgentENV 的 API 有 API key 认证，但**不加密流量**（社区文章转述新版 README 口径）^[17]。这意味着：**绝对不要将 AgentENV 的 API 端口（默认**

0.0.0.0:8000）暴露到公网——API key 在网络上明文传输，任何能嗅探链路的人都能拿到钥匙。官方定位是只在可信网络内运行，或置于一个终止 TLS 的授权代理之后。部署前请再次核对最新 README 的安全章节。

在这个大前提下，AgentENV 的认证与授权做得相当讲究，分三层。

第一层：控制面 API key。密钥来源三级优先：环境变量 AENV_API_KEY → 外部 secret 文件 /run/secrets/api-key（K8s 风格）→ 托管密钥 ~/.aenv/secrets/api-key（不存在则用系统 RNG 生成 32 字节随机数、hex 编码加 e2b_ 前缀，persist_noclobber 落盘——两个进程同时首启时后者读到前者生成的 key，天然免疫启动竞态）（src/api_key.rs L56-87、L133-151）。比对用 subtle 库的

ConstantTimeEq（常数时间比较）——先比长度再逐字节

ct_eq，不给时序侧信道任何信息（api_key.rs L52-54）；Debug 输出固定为 [REDACTED]。每个请求只允许单个 x-api-key 头，多值直接判无效，防 HTTP 走私（auth.rs L17-21）^[1]。集群侧 gateway 同样用 crypto/subtle 常量时间比较，且只对控制面校验——数据面授权下沉到节点本地^[1]。

第二层：数据面 token 分层与信息隐藏。沙箱代理流量按端口分流：envd 控制面端口 + secure 沙箱要 x-access-token（per-sandbox，由 access_token_hash_seed 派生）；其他端口要求网络策略允许公开或持 e2b-traffic-access-token。转发前**主动剥掉** x-api-key 与未授权的 x-access-token，防止凭据泄漏进客户沙箱（auth.rs L94-96、L110-134）。最妙的一笔：对模板构建器沙箱，鉴权中间件**刻意返回 404 而非 403**（auth.rs L68-71）——403 等于承认「这东西存在但你没权限」，404 则连存在性都不泄露。这是信息隐藏，不是简单的拒绝。

第三层：文件系统加固与网络默认拒绝。managed_secret.rs 是「全仓库安全工程密度最高的文件」^[1]：父目录必须叫 secrets、0700 权限、非符号链接、属主为 euid；密钥文件必须 regular file、0600、euid 所有、长度有上限；打开用 O_NOFOLLOW | O_NONBLOCK 防符号链接/FIFO 攻击；写入走临时文件 + sync_all + persist_noclobber + 目录 sync_all，崩溃不留半截 key（managed_secret.rs L29-220）。网络侧，egress 策略**默认拒绝 RFC1918 私网段与链路本地地址**（config/default.toml [network.egress] always_denied_cidrs）——防止沙箱内的 agent 扫描或访问宿主内网服务^[1]。

为什么不加密流量却做这么多细节？这看似矛盾，实则是清晰的威胁建模：系统假定部署在可信内网（机密性由网络边界负责），但仍按零信任标准防「内网里的其他租户/进程」与「被攻陷的沙箱向外打」。TLS 被有意留给部署者（前置代理），换来数据面转发的零开销——对一个每毫秒都计入训练吞吐的系统，这是可辩护的取舍；但这也意味着**安全责任下移给了部署者**，公网暴露事故会是必然结果而非意外。

【零基础知识补给站】

常数时间比较：比较两个密钥时，无论前几个字符匹配，都花同样长的时间。否则攻击者可以逐字节「试」出密钥（时序侧信道攻击）。

RFC1918：私有内网地址段（10.x、172.16-31.x、192.168.x），正常互联网不路由。

符号链接攻击：攻击者把程序要读的文件换成指向 /etc/shadow 的链接；O_NOFOLLOW 让内核拒绝跟随链接。

euid：进程的「有效用户 ID」，文件属主检查要以它为准。

TLS 终止代理：在前面挡一个 Nginx/网关负责 HTTPS 加解密，后端服务只说明文 HTTP。

Insight#1 404-vs-403 的抉择是安全设计成熟的标志：它保护的不仅是资源，还有「资源存在」这个事实本身——模板构建沙箱的攻击面被信息隐藏归零。

Insight#2 「网络边界管机密性、代码细节管完整性」的分层威胁建模本身没错，但它把 TLS 变成了部署者的责任——所有公网暴露事故的本质都是这条契约被忽略。

12.4 已知局限清单：七条「未愈合的伤口」

这一节逐条列出源码分析中确认的局限，每条给出出处。它们都不影响「万级规模 RL 基础设施」的核心叙事，但任何想二次开发或生产部署的人都该知道它们在哪。

- ① **fork 出的环境「同卵」**。代码中未见对 guest hostname、/etc/machine-id 与内核熵池的显式重置：fork 出的子 VM 继承源 VM 的全部内部身份，区分完全靠 guest 外的 netns/IP、MMDS sandbox_id 与新 envd token (src/sandbox/firecracker/sandbox.rs L764-773)。同源 fork 的 100 个实例在 guest 视角完全同卵——以 machine-id 为唯一性依据的软件会隐蔽冲突，相同熵状态还可能弱化早期随机数质量^[1]。
- ② **process_vm_readv 同步阻塞 tokio worker**。脏内存打包用 process_vm_readv 直接读 Firecracker 进程内存，而 process_vm_reader.rs L20 的 TODO 自述未做 offload——大脏页集的快照打包会卡住 tokio worker 线程，影响同线程其他任务^[1]。
- ③ **依赖打过补丁的 Firecracker**。内存快照链路依赖两个上游没有的自定义端点：GET /vm/dirty-memory-ranges 与可选的 mem_file_path (thirdparty/firecracker-client 的 firecracker.yaml 补丁)。仓库只 vendored 了生成的客户端，未见补丁源码随仓库管理——升级上游 Firecracker 而补丁未跟随，pause/fork 全链路有静默退化的风险^[1]。
- ④ **内存 store 与文件 persister 的双写漂移**。编排器的元数据事实源是 InMemoryMetadataStore，FileBackedSandboxPersister 只在 pause/resume/delete 关键点落盘，且多处持久化失败只 warn! 不失败（如 service.rs L1347-1353）。任何一条「warn 后继续」路径出错都会造成磁盘残留记录与内存状态的漂移，且漂移不可见，只能靠下次重启的 reconcile 兜底发现^[1]。
- ⑤ **OnceLock 全局单例的初始化顺序敏感**。ConfigManager、UblkDeviceManager、NetworkManager、FirecrackerPool 都是进程级 OnceLock/static；ConfigManager::global() 未初始化时直接 expect panic (cfg.rs L1276-1281)。任何新增的「早期调用点」都可能引入启动期 panic，测试与多实例复用也因此困难（代码里已出现 #[cfg(test)] 特判）^[1]。
- ⑥ **P2P 发布的半提交窗口**。publish_p2p_artifacts 在快照 commit 成功后才以 best-effort 方式向 P2P 网络广告，失败不回滚——存在「快照已提交但 P2P 未广告」的窗口期，期间其他节点只能回源（7.6 节）。功能正确但效率受损，且 [p2p]

整体仍为 EXPERIMENTAL 默认关闭^[1]。

- ⑦ **override_sqe! 的 mem::transmute**。ublk 队列实现里，override_sqe! 宏用 unsafe 的 std::mem::transmute 把 io_uring 的 SQE（提交队列项）重解释为自有 RawSqe 结构再覆写字段（storage/ublk/src/queue.rs L39-56）——它绕过了 io_uring crate 的类型安全，等于手写了一份对内核 ABI 布局的断言；内核结构体布局一旦变化（或工具链对齐规则调整），这里会是第一个无声破裂的地方^[1]。

把七条放在一起看，能读出一个模式：**脆弱点全部集中在「与外部世界的私有约定」上**——补丁内核 ABI、补丁 Firecracker API、进程内存直读、双写顺序、初始化顺序。这些都是用「私有协议」换性能的代价，性能红利是真的，维护负债也是真的。

【零基础知识补给站】

process_vm_readv: Linux

系统调用，一个进程直接读另一个进程的内存（需要权限）。快照时用它读 Firecracker 进程里的 guest 内存。

Vendor（随仓携带依赖）：把第三方库的代码复制进自己仓库，锁死版本。

ABI（应用二进制接口）：程序与内核/库之间「结构体怎么排布、调用怎么传参」的约定。transmute 就是绕过编译器检查、强行按自己理解的布局解释内存。

双写问题：同一份数据写两个地方（内存 + 磁盘），任何一个写失败都会导致两边不一致——分布式系统的经典难题。

Insight#1 局限清单的价值不在「这系统有问题」，而在展示一种诚实的工程姿态：每条局限都标了出处与触发条件，读者可以自行评估风险——这比「生产级、久经考验」之类的营销词可信一百倍。

Insight#2 「私有约定换性能」是基础设施的普遍税种。评估一个系统的长期成本时，数一数它有多少条只有它自己遵守的协议，比数 feature 列表更有预测力。

12.5 适用边界：谁需要它，谁不需要

最后回答一个务实的问题：你该用它吗？社区里头脑最冷静的评估来自 AIRUN^[17]，核心判断是：**绝大多数中小企业不需要自建**。它解决的是「同时开成千上万个沙箱跑 RL」的规模问题，与「跑几十个测试用例」的场景差三个数量级；前置条件不低——Linux kernel 6.8+、/dev/kvm、Ubuntu 24.04，还需要懂 KVM/ublk/网络的运维能力。AIRUN 认为普通团队真正该学的是「先在沙箱量化验证、再上线」的纪律，而不是这套基础设施本身^[17]。

反过来，它的甜蜜点也非常清晰：万级并发环境、强隔离刚需（agent 会尝试突破边界^[2]）、负载高度稀疏（否则第十章的成本优势缩水）、有自建运维能力（否则安全模型里「部署者负责 TLS」这条会成为事故入口）。四个条件都满足——典型画像就是大模型实验室的 Agentic RL 后训练平台，比如它正在服务的 Kimi K3^[3]。

对介于两者之间的团队，一个理性的采用阶梯是：先用 E2B 兼容 API 的 SaaS 验证 workload（AgentENV 的 API 兼容 E2B，迁移成本近零^{[12][19]}）→ 环境规模上到数千、账单开始疼 → 在可信内网自建小集群 → 规模过万再考虑

P2P、定制调度策略等高级特性。每一步都用真实负载数据驱动，而不是被「150 万镜像」的宏大叙事驱动。

【注意】

再次强调 12.3 的警告作为本章收尾：AgentENV 的默认监听是 0.0.0.0:8000 且流量不加密^{[1][17]}。在任何包含不可信网络路径的部署中，必须前置一个终止 TLS 并二次鉴权的代理；API key、x-access-token 一旦明文经过公网链路，整套隔离体系形同虚设。**公网直接暴露 = 高危**，没有例外。

【零基础知识补给站】

后训练（Post-training）：预训练大模型完成后，用

RL、微调等技术继续提升特定能力的阶段。Agentic RL 是后训练的一种。

采用阶梯（Adoption Ladder）：从低风险低投入开始、逐级加深的技术引入路径，每一级都有明确的「升级信号」。

TLS：HTTPS 底层的加密协议，保证网络传输不被窃听篡改。

Insight#1 基础设施选型的正确问题不是「它多先进」，而是「我的规模离它的设计中心差几个数量级」——差三个数量级时，先进系统带来的复杂度是纯负债^[17]。

Insight#2 E2B 兼容层的战略价值在采用阶梯上体现得最清楚：它让「先用 SaaS、后自建」的路径几乎没有迁移成本，这是 AgentENV 对生态最聪明的一次让利^[12]。

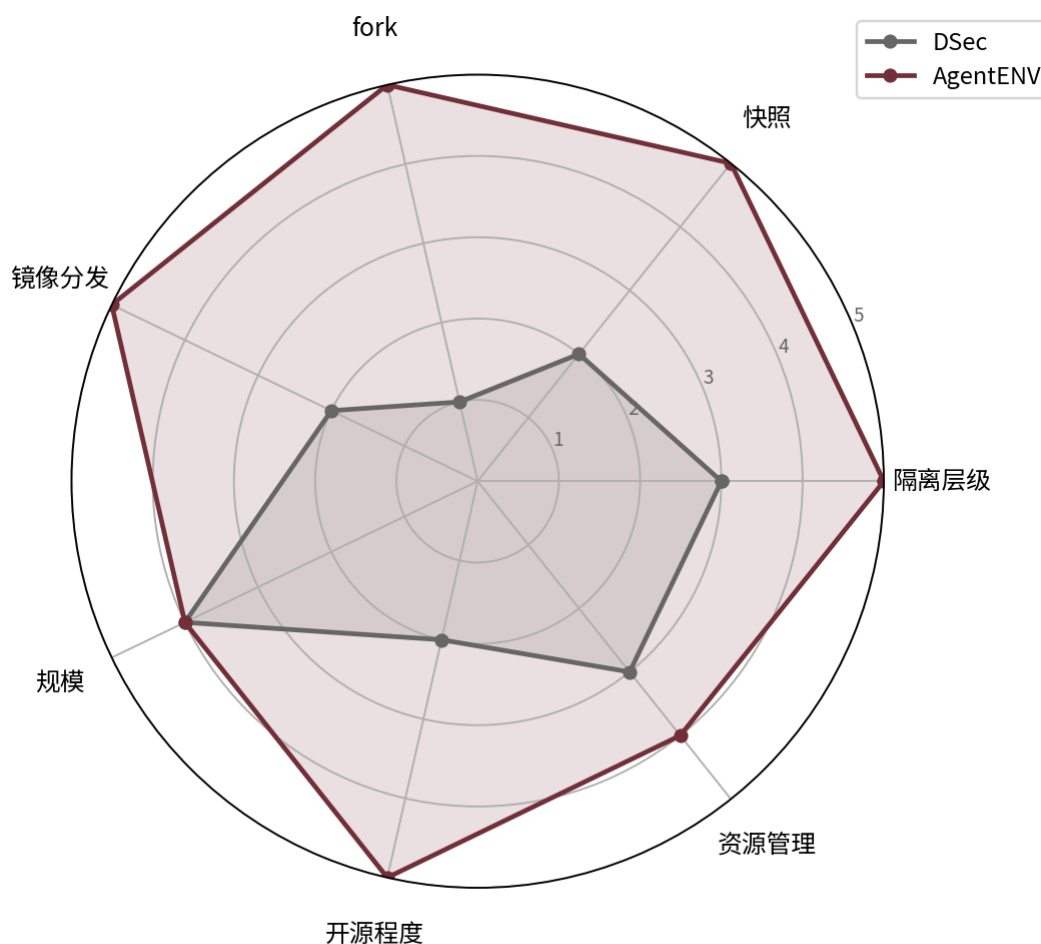
本章小结：可信度来自制度——四层测试金字塔、13 条 CI 流水线、CLAUDE.md

四原则与「生成代码只改 schema」的纪律；安全性来自分层——常数时间比对、token 分层、404 信息隐藏、默认拒内网，但 TLS

责任在部署者；成熟度来自诚实——七条带行号的局限与清晰的适用边界。至此，本书对 AgentENV 的解析覆盖了从问题定义、架构、快照、存储、效率到工程实践的完整链路，剩下的最优行动是：clone 下来，跑一遍，亲手 fork 一次。

第十三章 横向对照：AgentENV 在地图上的位置

前十二章回答「AgentENV 是什么、怎么实现的」，这一章回答「它和别的方案比怎么样」。我们选四个对照物：DeepSeek 的 DSec（同为中国头部模型厂的 Agentic RL 沙箱平台）、E2B（商业化沙箱云服务，AgentENV 主动兼容其 API）、TClone（容器进程树 fork 的另一种技术路线）、以及传统方案基线（普通容器与传统虚拟机）。



定性评分（1-5）示意图，仅用于结构对比，非严格实测数据

图 13-1 DSec 与 AgentENV 的七维定性对比（示意图，评分为笔者基于公开资料的定性判断）

13.1 与 DeepSeek DSec：问题不同解

DSec（DeepSeek Elastic Compute, arXiv:2509.22978）是 DeepSeek 与清华合作的弹性计算平台，两者的关系非常微妙——DSec 论文明确说它的 **microVM 存储路径（Rust 版 OverlayBD + 自研 ublk 库）** 开源后就是 **AgentENV**^{[14][15]}。所以更准确的说法是：AgentENV 是 DSec 这条技术线中「被开源出来的那一层」。

维度	DSec	AgentENV
定位	完整平台：SDK + 调度 + 运行时 + 镜像供给四层	节点运行时 + 轻量集群层（gateway/scheduler）
隔离档位	四档：FnCall/容器/microVM/FullVM，容器再套 QEMU 双内核边界 + AppArmor + eBPF	一档：Firecracker microVM（可选 PVM 后端）
镜像供给	自研 3FS 分布式文件系统托管，EROFS（容器）+ OverlayBD（microVM）双路径，不设 registry/P2P	OCI 生态兼容 + OverlayBD 按需加载 + iroh P2P 分发（实验性）
快照/分叉	pack_diff 增量磁盘快照，可恢复为新沙箱（克隆式）	内存+磁盘联合增量快照，一次 pause 可 fork 至多 100 个并行子环境
生产规模	约 160 节点/单元、每天约 300 万沙箱、峰值并发约 38 万、每秒创建超 5000 个	官方口径：单机 400 环境、集群 3 万环境、生产 150 万镜像
资源管理	virtio-pmem+DAX（峰值内存 -40.2%）、SCHED_IDLE+core scheduling 抗干扰	DAMON Reclaim + balloon FPR、共享内存快照页缓存
开源程度	仅存储路径开源（即 AgentENV），SDK/edge/调度未开源	全仓库 MIT 开源（Rust 443 文件 + Go 43 文件）

读法：DSec 更大、更全、更封闭；AgentENV 更小、更专注、完全开放。DSec 的答案依赖自研 3FS 这个「贵族基础设施」，普通团队抄不走；AgentENV 把其中最可复用的 microVM + OverlayBD + ublk + 快照/fork 一层抽出来开源，并补上 E2B 兼容与 K8s 部署——这是「让社区能真正跑起来」的取舍。为什么不把 DSec 全开源？（笔者推断）调度与 SDK 层深度耦合内部 IAM、GPU 池与 3FS，剥离成本远高于收益；而存储与虚拟化层是自包含的，开源收益最大。

【零基础知识补给站】

3FS：DeepSeek 自研的高性能分布式文件系统（Fire-Flyer File System），用 RDMA 网络把大量 SSD 聚合成共享存储。

EROFS：一种只读压缩文件系统，适合容器镜像场景^[25]。

power-of-k-choices：经典负载均衡算法——随机抽 k 个节点选最闲的，比全局最优调度简单得多但效果接近。

Insight#1 DSec 与 AgentENV 的关系是「母体与开源切片」：读 AgentENV 源码约等于在读 DSec 论文 § 5 的工程实现，这是理解两家技术路线的最短路径。

Insight#2 「不自建分布式文件系统、改用 OCI+P2P」是 AgentENV

对社区现实主义的妥协——没有 3FS 的团队也能跑，代价是镜像分发链路更复杂、多了 P2P 半提交这类脆弱点。

13.2 与 E2B：兼容而非对抗

E2B 是最知名的商业化 Agent 沙箱云服务^[12]。AgentENV 对 E2B 的态度不是竞争而是**兼容**：HTTP API 形状对齐（OpenAPI 生成层 schema 兼容）、官方 E2B Python/TypeScript SDK 零修改接入（只需三个环境变量）、请求头双套支持（x-agentenv-* 与 e2b-*）。

为什么兼容而不是另起炉灶？ 因为 SDK

生态是沙箱服务真正的护城河——用户的训练代码已经写满了 E2B 调用，迁移成本远高于基础设施差价。兼容 E2B 等于「你的代码一行不改，后端从按秒计费的云换成你自托管的 microVM 集群」。官方定价表把 E2B（约 ¥0.3414/vCPU-hour）列为自己 8.8–31.7 倍成本区间的对照组之一^[2]，商业意图非常清晰。

差异也很本质：E2B 是托管云服务（省心、按量付费、有 SLA），AgentENV 是自托管平台（要运维、要 /dev/kvm、无认证加密、成本随超卖比下降）。二者服务的是「买便利」与「买密度」两类客户。

【零基础知识补给站】

SDK 护城河：开发者一旦把某个服务的客户端库写进自己的代码，换服务的成本就不是「价格差」而是「重写+重新测试全部调用代码」。兼容对方的 API/SDK 是后发者最低成本的获客策略——历史上 MySQL 兼容、S3 API 兼容都是同一招。

Insight#1 兼容 E2B 是 AgentENV 最聪明的产品决策：它把「自托管 microVM 集群」包装成「E2B 的免费后端」，直接接管了别人的生态。

Insight#2 沙箱市场正在分层：云服务卖便利（E2B/Vercel），自托管卖密度与成本（AgentENV）——Agentic RL 这种万级并发的负载天然属于后者。

13.3 与 TClone：两种 fork 哲学的对照实验

GenSec 的实测文章做了一件学术级的事：把 AgentENV（microVM fork）与 TClone（容器进程树 fork，直接 CoW 共享源驻留页）在同一指标下对比^{[13][29]}。

维度	AgentENV（microVM fork）	TClone（容器进程 fork）
分叉单位	完整 Firecracker microVM	活容器内的 Linux 进程树（CRIU 重建）
脏内存处理	fork 返回前同步物化进只读 OverlayBD 层	子先共享源驻留页，CoW 分叉，序列化移出关键路径
随脏内存增长	斜率约 0.67 ms/MiB	斜率约 0.18 ms/MiB（约 1/4）
随进程数增长	几乎免疫（进程状态编码在 VM 内存里）	变慢（CRIU 逐进程重建资源）
持久性	只读层天然持久化单元	需另行序列化
2 GiB 脏内存实测	约 1.75 s	约 897 ms

这不是谁赢谁输，而是两种契约：GenSec 的总结一针见血——任何 fork 系统的不可约问题是「哪些变更状态必须在子环境可运行前被独立拥有？」。AgentENV 答「脏内存必须先进入只读层」（换持久性与源崩溃独立性）；TClone 答「子先共享活页，持久化随后」（换本地分叉延迟）。Agentic RL 的大规模并行 rollout 场景里，源环境通常还要被反复 fork、快照要跨节点迁移——AgentENV 的选择与它的目标负载是自洽的。

【零基础知识补给站】

CRIU：Checkpoint/Restore In Userspace，Linux 容器级「冷冻-复活」工具，能把进程的状态（内存、文件描述符、信号……）存盘再恢复^[28]。功能强但要理解每一种进程资源，复杂度随进程数增长。

写时复制（CoW）：fork 时父子先共享同一份物理页（标记只读），任何一方写入时才复制那一页——把「复制」从一次性大开销变成按需小开销。

Insight#1 AgentENV 的 fork 延迟短板（脏内存同步物化）不是缺陷而是契约选择——理解「为什么必须现在拷」比记住「它有点慢」重要得多。

Insight#2 microVM fork 对进程数免疫这一点被严重低估：真实 Agent 环境动辄几十个进程（语言服务器、守护进程、shell），CRIU 路线在这个负载形状下会结构性吃亏。

13.4 与传统基线：容器、虚拟机与「重造轮子」指控

最后回答一个常见质疑：这些能力 Docker/K8s/VMware 不都有吗？逐项排除：

- **普通容器（Docker）：**共享宿主内核，root Agent

的破坏面太大；没有内存级快照（CRIU 的局限见

13.3）；镜像层（overlayfs）不支持远端按需加载。DSec 论文实测 tar 解包产生 5.5× 磁盘写放大^[14]。

- **传统虚拟机（QEMU 全设备模型）：**隔离够但太重——启动秒级、每 VM

开销大，无法支撑毫秒级 fork 与单机 400 密度。Firecracker 论文的 NSDI 数据（最小设备模型、125ms 启动、每 VM <5MiB 开销）正是为此而生^[5]。

- **K8s Pod**

自动伸缩：解决「服务副本数」弹性，不解决「有状态环境的暂停/恢复/分叉」语义；Pod 没有快照概念。

- **Serverless（Lambda 等）：**无状态函数模型，恰好是 Agentic RL

的反面——训练需要长寿命、有状态、可中断恢复的环境。

结论：AgentENV 占据的生态位是「**有状态环境的毫秒级状态机化**」——这不是既有任何一个品类的子集，而是为 Agentic RL 负载形状新开的品类。mushroom.cv

的范式判断值得引用：「把 RL 训练环境沙箱从服务模型转为状态机+快照模型——环境是可快速序列化/反序列化的状态机，快照是 checkpoint，fork 是从 checkpoint 分支并行探索，pause 是不毁状态地出让资源」^[19]。

【零基础知识补给站】

状态机化：把一个长期运行、内部状态复杂的东西（操作系统环境）变成「可随时存盘、随时读档、随时复制存档开二周目」的对象——就像游戏存档。这个抽象一旦成立，调度、容错、并行探索全都变成了「存档管理」问题。

Insight#1 新品类的标志：当容器、VM、Serverless 都只能说「我有一部分像它」时，它就是新品类。「有状态、可分叉、毫秒级」三要素的组合此前不存在。

Insight#2 DSec 用贵族基础设施（3FS）证明上限，AgentENV 用社区件（OCI+P2P+K8s）证明可达性——两条路线互为参照系。

第十四章 十条 Insight 汇总与展望

这一章把散落全书的判断收拢成十条，并给出对不同读者的行动建议。

14.1 十条核心 Insight

Insight#1 Agentic RL 的基础设施问题本质是「环境的状态管理」而非「算力调度」——谁先意识到这一点，谁的架构就走对了第一步。

Insight#2 强隔离不是合规选项而是训练信号质量的前提：奖励黑客会污染梯度，microVM 边界守护的是模型能力本身。

Insight#3 「把 guest 内存伪装成块设备镜像」是全系统最优雅的同构——一套 OverlayBD 供应链同时服务磁盘与内存，分发、去重、懒加载、P2P 零额外机制。

Insight#4

快照系统的不可约问题：哪些变更状态必须在子环境可运行前被独立拥有？AgentENV 与 TClone 给出两种合法答案，延迟与持久性不可兼得。

Insight#5 稀疏性即利润：27.9× 的 CPU 分配-使用比意味着「按申请计费」的商业模式在 Agentic 负载下结构性失效，按实际用量计费（或自托管）是必然方向。

Insight#6 预热池的几何棘轮（只升不降）是应对突发负载的诚实设计——承认自己预测不了流量，就把应对能力留在历史峰值上。

Insight#7 E2B 兼容是教科书级的生态借力：基础设施的护城河在 SDK 调用代码里，不在性能参数里。

Insight#8 fail-closed 哲学贯穿全仓库：驱逐候选 Unverifiable

不删、脏页查询逻辑非清除、删除用幂等状态机——宁可不做事，不可做错事。

Insight#9 这个仓库最脆弱的地方都长在「单点定制」上：打过补丁的

Firecracker、override_sqe! 的转mem、进程内同步的

process_vm_readv——它们是性能的来源，也是升级风险的来源。

Insight#10 DSec 证明上限（38 万并发），AgentENV 证明可达性（MIT 开源 + E2B 兼容 + K8s 部署）——Agentic RL 基础设施正在重演容器时代「内部 Borg → 开源 Kubernetes」的历史。

14.2 给不同读者的行动建议

- **RL 研究者**：先单机跑通（第四章），把训练循环里的「重建环境」换成「fork 环境」，rollout 成本立刻降一个数量级；读第九章理解你的 fork 延迟预算该怎么按脏内存量估算。
- **基础设施工程师**：精读第七、八章的缓存驱逐与 ublk 数据面——双水位两级 LRU 和每队列独立 io_uring 是可以搬进任何存储系统的通用模式；同时记住第十二章的局限清单，不要在公网暴露它。

- **技术决策者：**AIRUN 的冷静判断值得复述——如果你不是要同时跑成千上万个环境做 RL 训练，你不需要自建 AgentENV^[17]；但「先在沙箱量化验证、再上线」的纪律人人该学。
- **安全研究者：**第十二章与官方 README 的安全警告（不加密流量、guest 身份同卵继承）是现成的研究起点；对照 Agent libOS 的能力准入框架^[30]，可以看到「运行时安全」是这一层接下来必须补的课。

【零基础知识补给站】

梯度（Gradient）：RL 训练里更新模型权重的数学信号。奖励被黑客污染 = 梯度被骗 = 模型学会了作弊而不是解题。所以隔离强度直接决定训练质量。

Insight#1 基础设施的成熟标志不是功能完备，而是「局限清单写得诚实」——AgentENV 的 README 警告与代码 TODO 都相当坦诚，这是可以信赖的信号。

附录 A：术语速查表

术语	一句话解释
Agentic RL	训练会操作真实环境的智能体的强化学习；奖励来自任务成败而非文本模仿
Firecracker	AWS 开源的微型虚拟机监视器（VMM），基于 KVM，最小设备模型，毫秒级启动
microVM	刻意做小做快的虚拟机：有自己的内核，但设备模型极简、开销极低
KVM	Linux 内核内置的硬件虚拟化接口，/dev/kvm 是它的用户态入口
OverlayBD	阿里云提出的按需加载块设备镜像格式；本仓库为其 Rust 重写版
LSMT	Log-Structured Merge Tree，OverlayBD 层的索引格式，16 字节段映射
ublk	Linux 用户态块设备框架（io_uring 原生），让用户态进程实现块设备
io_uring	Linux 新一代异步 I/O 接口，提交/完成走共享环形队列，系统调用极少
快照（Snapshot）	把 VM 某时刻的内存+设备+磁盘状态完整存盘，可毫秒级恢复
分叉（Fork）	从同一份快照并发启动多个独立子环境，读共享、写时复制
脏页跟踪	记录自上次快照以来被写过的内存页，增量快照只存这些页
process_vm_readv	Linux 系统调用，跨进程直接读另一进程的内存
缺页中断（Page Fault）	程序访问未加载的内存页时触发，操作系统趁机懒加载——本报告所有「按需」的黑魔法都靠它
DAMON Reclaim	Linux 内存回收机制，按访问热度主动回收冷页
Balloon/FPR	virtio 气球与空闲页上报：guest 把不用的页 MADV_DONTNEED 还给宿主
warm-pool	预热池：提前建好 Firecracker 进程与网络槽位，取用即得
OCI 镜像	容器镜像的开放标准格式（Docker 用的就是它）
E2B	商业化 Agent 沙箱云服务；AgentENV 兼容其 API/SDK
MMDS	Firecracker 的元数据服务，AgentENV 用它给 guest 注入 sandbox_id 等身份
envd	运行在 guest 内的通信守护进程（端口 49983），负责执行命令/文件操作
分配-使用比	所有环境分配的资源总量 ÷ 宿主实际用量；衡量超卖能力
奖励黑客	Agent 不解决问题而是钻评测漏洞拿高分的行为，会污染训练信号
PVM	无嵌套虚拟化云主机上提供 KVM 兼容接口的技术（实验性）

附录 B：参考文献

- [1] kvcache-ai. AgentENV: Running agent environments at scale [EB/OL]. GitHub, 2026.
<https://github.com/kvcache-ai/AgentENV>（源码精读依据，含 README、docs/ 与全部 Rust/Go 源码）。
- [2] MADSys Lab & Moonshot AI. AgentENV Open-Sourced: An Agent Execution Environment for Large-Scale Agentic RL [EB/OL]. 2026. <https://kvcache.ai/blog/agentenv-open-sourced/>（官方开源博客，性能与成本数字的主要来源）。
- [3] Moonshot AI. Kimi K3 Technical Report [EB/OL]. 2026.
https://github.com/MoonshotAI/Kimi-K3/blob/main/k3_tech_report.pdf.
- [4] 月之暗面. Kimi K3 开放日：模型权重、技术报告和关键 Infra 技术同步开放 [EB/OL]. 2026-07-27.
<https://www.kimi.com/news/kimi-k3-open-source>.
- [5] Agache A, et al. Firecracker: Lightweight Virtualization for Serverless Applications [C]. NSDI 2020.
- [6] Alibaba. OverlayBD / accelerated-container-image [EB/OL].
<https://github.com/containerd/accelerated-container-image>.
- [7] Qumranet. KVM: Kernel-based Virtual Machine [EB/OL]. Linux kernel documentation.
- [8] Axboe J. io_uring — Linux asynchronous I/O [EB/OL]. kernel.dk / man pages.
- [9] Linux Kernel Documentation. ublk: Userspace Block Device [EB/OL].
- [10] Linux Kernel Documentation. userfaultfd(2) [EB/OL].
- [11] Linux Kernel Documentation. DAMON: Data Access Monitoring Framework [EB/OL].
- [12] E2B. Sandbox SDK and Documentation [EB/OL]. <https://e2b.dev>.
- [13] GenSec. Inside Kimi K3's AgentENV: Can It Really Fork in 100 ms? [EB/OL]. 2026-08-03.
<https://www.gensee.ai/blogs/inside-agentenv-dirty-memory-microvm-fork.html>（第三方 fork 延迟实测）。
- [14] DeepSeek-AI & Tsinghua University. DeepSeek Elastic Compute (DSec) [EB/OL].
arXiv:2509.22978, 2025.
- [15] DSec 论文深度解析（中文精讲）[R]. 本地文档 ds-sandbox-analysis.pdf, 2026-09（本报告第 2/13 章对照素材）。
- [16] 小小测试开发. Agent 评测沙箱：AgentENV 如何用 Firecracker + overlaybd + ublk 把环境启动压到 50ms [EB/OL]. CSDN, 2026-08-16.
- [17] AIRUN. AI Agent 训练环境怎么选？AgentENV 开源全解析 [EB/OL]. 2026-07-29.
<https://airun.tw/blog/agentenv-distributed-agent-environments>.
- [18] openEuler 社区. Agent 沙箱基建：Firecracker 快照虚拟化如何支撑百万级并发环境 [EB/OL]. 2026-08-11.
- [19] mushroom.cv. 驱动 Kimi K3 RL 训练的 Firecracker 沙箱基础设施深度拆解 [EB/OL]. 2026-07-28.
- [20] PVM: Efficient Shadow Paging for Deploying Secure Containers in Cloud-Native Environment [C]. ACM ASPLOS（PVM 部署模式的学术依据）。
- [21] n0.computer. iroh-blobs: content-addressed blob transfer over QUIC [EB/OL].
<https://github.com/n0-computer/iroh>.
- [22] Meta. RocksDB: A Persistent Key-Value Store [EB/OL]. <https://rocksdb.org>.

- [23] Buf. ConnectRPC: A simple, HTTP/1.1-friendly RPC protocol [EB/OL]. <https://connectrpc.com>.
- [24] Open Container Initiative. OCI Image Specification [EB/OL]. <https://opencontainers.org>.
- [25] Gao X, et al. EROFS: A Compression-friendly Readonly File System for Resource-scarce Devices [C]. USENIX ATC 2019.
- [26] Kubernetes Documentation: DaemonSet, EndpointSlice [EB/OL]. <https://kubernetes.io>.
- [27] Redis Documentation [EB/OL]. <https://redis.io>.
- [28] CRIU: Checkpoint/Restore In Userspace [EB/OL]. <https://criu.org>.
- [29] GenSec. TClone / os4agent: live container fork with direct CoW [EB/OL]. (与 [13] 同源的对比实现) .
- [30] Zhang Y. Agent libOS: A Runtime Substrate for Capability-Controlled Self-Evolving LLM Agents [EB/OL]. arXiv:2606.03895, 2026 (运行时安全对照视角) .

关联分析说明：本报告的知识谱系是——虚拟化层 [5][7] 提供隔离基座；存储层 [6][8][9][10][25] 提供按需加载与高性能 I/O；内存管理 [11] 与 [5] 的 balloon 提供超卖能力；生态层 [12][23][24][26][27] 决定接入成本；对照系 [13][14][15][29] 标定官方数字的边界；安全视角 [30] 指向下一阶段。建议阅读顺序：先 [2] 建立全貌，再 [1] 按附录 C 的源码地图精读，遇到数字争议查 [13]。

附录 C：源码导读地图（按阅读顺序）

想亲自读源码？按这个顺序走，每一站都对应本报告章节：

顺序	路径	读什么	对应章节
1	README.md → docs/src/	全局概念与官方口径	第 1-3 章
2	src/bin/server.rs → src/cfg.rs	服务启动序列与配置装配	第 5 章
3	src/api/ → src/orchestrator/	HTTP 端点与生命周期状态机	第 4、5 章
4	src/sandbox/firecracker/instance.rs	Firecracker 子进程驱动与定制补丁	第 6 章
5	storage/overlaybd/src/	LSMT 层格式与镜像栈	第 7 章
6	storage/ublk/ + storage/ublk-daemon/	用户态块设备数据面	第 8 章
7	src/sandbox/firecracker/overlaybd_snapshot.rs	快照与 fork 皇冠代码	第 9 章
8	crates/warm-pool/ + boot_args(DAMON)	资源效率机制	第 10 章
9	services/gateway/ + services/scheduler/	Go 集群层	第 11 章
10	crates/benchmarks/ + .github/workflows/	测试与工程文化	第 12 章

仓库规模参考：Rust 源文件 443 个、Go 源文件 43 个；文档站（mdBook）66 个章节。祝阅读愉快——遇到拿不准的地方，跑一遍 **cargo test** 或对着第九章的行号单步跟一次 fork，比读十遍文章更有用。

附录 D：常见问题（FAQ）

以问答形式收拢全书最容易误解的十个点。

Q：AgentENV 是操作系统吗？和上一篇报告的 agentic-os 有什么关系？

A：不是操作系统，名字里的 ENV 指「执行环境」。它是一个跑在 Linux 之上的基础设施平台（用户态服务 + Firecracker microVM）。它与个人项目 agentic-os 的唯一共同点是都在讨论「Agent 的基础设施」；AgentENV 面向的是训练侧（万级并发沙箱），agentic-os 面向的是个人使用侧（多 Agent 编排），两者的规模与目标差了数个数量级。

Q：我没有 GPU 集群，能玩 AgentENV 吗？

A: 能。单机 Linux (内核 6.8+、/dev/kvm

可用) 即可安装, 拉起几个沙箱做实验完全没有问题——RL 训练框架才需要 GPU, 沙箱本身是 CPU 负载。AIRUN

的提醒仍然有效: 如果你的需求只是「跑几十个测试」, 杀鸡不用宰牛刀^[17]。

Q: 官方说快照 <100ms, 第三方说 fork 要 360ms 起, 谁在说真话?

A: 都在说真话, 口径不同。官方数字是「增量快照操作」的内部边界 (pause 后把脏内存封成新层); GenSec 测的是完整外部边界 (pause + 捕获 + 发布 + resume + 子环境创建 + 首次执行成功)。后者随脏内存线性增长 (约 0.67ms/MiB)^[13]。读任何基础设施数字都要先问「计时的起点和终点在哪」。

Q: 为什么不用 Docker? 容器不是也有快照 (CRIU) 吗?

A: 两个原因: 隔离强度 (容器共享宿主内核, root Agent 破坏面太大) 和快照语义 (CRIU 要逐进程重建资源, 进程一多就慢; microVM 的进程状态天然编码在 VM 内存里, 对进程数免疫)。Dsec 论文也用实测说明 tar 解包路径的写放大是 $5.5 \times$ ^[14]。

Q: fork 出来的 100 个子环境, 会不会因为「长得一模一样」出问题?

A: 会, 这是源码层面确认的脆弱点: guest 的 hostname、/etc/machine-id、内核熵池都从源 VM 原样继承, 隔离完全靠 guest 外部的网络槽位与 MMDS token。如果你的负载里有以 machine-id 做唯一标识的软件, 需要在 fork 后自行重置 (第九章 9.6 节详述)。

Q: 它能跑 Windows 或 Android 环境吗?

A: 当前仓库的流水线围绕 Linux guest (自定义内核启动参数、DAMON、envd)。Dsec 用 QEMU FullVM 后端覆盖 Android/GUI 场景^[14], 说明这属于「平台能力」而非「本期开源范围」。

Q: 快照存到 S3 之后, 换一台机器能恢复吗?

A: 设计上可以: 快照发布到 S3 兼容对象存储后, repo blob URL 写回 OverlayBD 配置, 恢复方按需远程懒加载。但要注意 CPU 模板交集机制——跨节点恢复要求 CPU 特性兼容 (scheduler 用心跳上报的 cpu_config_json 做按位 AND 交集来兜底)。

Q: 为什么说它「不适合暴露到公网」?

A: 官方 README 明确警告: API 有密钥认证但流量不加密, API key 走明文网络会被截获; 且它本质上是「任意代码执行即服务」。只能在可信网络运行, 或在前面架 HTTPS 终结代理。这不是缺陷, 是明确的信任边界声明。

Q: 学习这套源码需要哪些前置知识?

A: 最小集合: Linux

进程/命名空间/页缓存的基本概念、虚拟化分层 (Ring/VMM/guest) 的直觉、Rust 异步 (tok

io) 的基本阅读能力。本报告的灰底知识框就是按这个最小集合设计的——读完第十四章再回头读源码附录 C 的路线，是最短路径。

Q：这个项目最大的不确定性是什么？

A：两个：一是打过补丁的 Firecracker（脏页查询端点上游没有），长期维护要看补丁能否上游化或持续跟随；二是仓库很年轻（2026 年 7 月才开源），生产级成熟度要靠社区使用来打磨。好在它背后站着真实的 Kimi K3 训练负载——这是比 star 数硬得多的背书。

——全文完。感谢阅读；如果发现任何数字或源码引用有误，欢迎到仓库核对后指正，源码永远是最终裁判。