openEuler 26.09 技术白皮书

1. 概述

OpenAtom openEuler（简称“openEuler”） 社区是一个面向数字基础设施操作系统的开源社区。由开放原子开源基金会（以下简称“基金会”）孵化及运营。

openEuler 是一个面向数字基础设施的操作系统，支持服务器、云计算、边缘计算、嵌入式等应用场景，支持多样性计算，致力于提供安全、稳定、易用的操作系统。通过为应用提供确定性保障能力，支持 OT 领域应用及 OT 与 ICT 的融合。

openEuler 社区通过开放的社区形式与全球的开发者共同构建一个开放、多元和架构包容的软件生态体系，孵化支持多种处理器架构、覆盖数字基础设施全场景，推动企业数字基础设施软硬件、应用生态繁荣发展。

2019 年 12 月 31 日，面向多样性计算的操作系统开源社区 openEuler 正式成立。

2020 年 9 月 30 日，首个 openEuler 20.09 创新版发布，该版本是 openEuler 社区中的多个企业、团队、独立开发者协同开发的成果，在 openEuler 社区的发展进程中具有里程碑式的意义，也是中国开源历史上的标志性事件。

2021 年 3 月 31 日，发布 openEuler 21.03 内核创新版，该版本将内核升级到 5.10，并在内核方向实现内核热升级、内存分级扩展等多个创新特性，加速提升多核性能，构筑千核运算能力。

2021 年 9 月 30 日，全新 openEuler 21.09 创新版如期而至，这是 openEuler 全新发布后的第一个社区版本，实现了全场景支持。增强服务器和云计算的特性，发布面向云原生的业务混部 CPU 调度算法、容器化操作系统 KubeOS 等关键技术；同时发布边缘和嵌入式版本。

2022 年 3 月 30 日，基于统一的 5.10 内核，发布面向服务器、云计算、边缘计算、嵌入式的全场景 openEuler 22.03 LTS 版本，聚焦算力释放，持续提升资源利用率，打造全场景协同的数字基础设施操作系统。

2022 年 9 月 30 日，发布 openEuler 22.09 创新版本，持续补齐全场景的支持。

2023 年 3 月 30 日，发布 openEuler 23.03 内核创新版本，采用 Linux Kernel 6.1 内核，为未来 openEuler 长生命周期版本采用 6.x 内核提前进行技术探索，方便开发者进行硬件适配、基础技术创新及上层应用创新。

2023 年 9 月 30 日，发布 openEuler 23.09 创新版本，是基于 6.4 内核的创新版本（参见版本生命周期），提供更多新特性和功能，给开发者和用户带来全新的体验，服务更多的领域和更多的用户。

2024 年 9 月 30 日，发布 openEuler 24.09，基于 6.6 内核的创新版本，提供更多新特性和功能。

2025 年 3 月 30 日，发布 openEuler 25.03 是基于 6.6 内核的创新版本，面向服务器、云、边缘计算和嵌入式场景，提供更多新特性和功能，给开发者和用户带来全新的体验，服务更多的领域和更多的用户。

2025 年 9 月 30 日，发布 openEuler 25.09，基于 6.6 内核创新版本，面向服务器、云、AI 和嵌入式场景，持续提供更多新特性和功能扩展，包括内核优化、异构协同推理、众核高密、机密容器、机密虚机、多核多实例混部、编译器、可信计算、密码加速等，给开发者和用户带来全新的体验，服务更多的领域和更多的用户。

2026 年 9 月 30 日，发布 openEuler 26.09，基于 6.6+ 6.18 的双内核创新版本，面向 AI 等场景基于 6.6 内核持续提供更多新特性和功能扩展，包括 Agent 可观测、SkillHub、上下文管理、KV Cache 协同加速、模型加载优化等，同时基于 6.18 内核增强 DevStation 智能开发者桌面能力，给开发者和用户带来全新的体验，服务更多的领域和更多的用户。

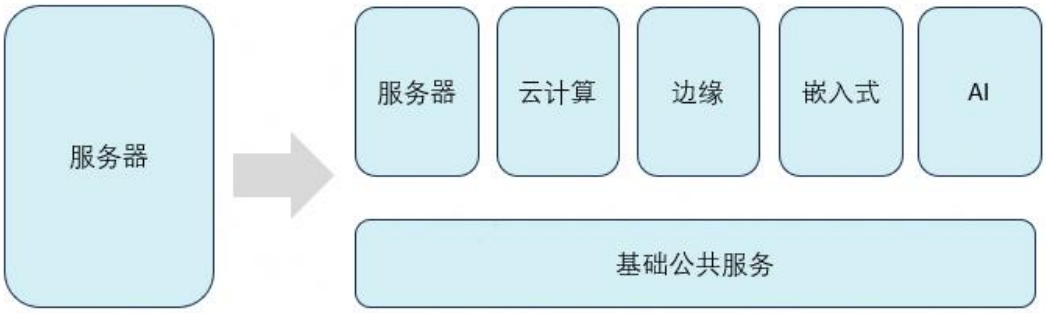


openEuler 作为一个操作系统发行版平台，其 LTS 版本为企业级用户提供一个安全稳定可靠的操作系统。

openEuler 也是一个技术孵化器。通过发布创新版，快速集成 openEuler 以及其他社区的最新技术成果，将社区验证成熟的特性逐步回合到发行版中。这些新特性以单个开源项目的方式存在于社区，方便开发者获得源代码，也方便其他开源社区使用。

社区中的最新技术成果持续合入社区发行版，社区发行版通过用户反馈反哺技术，激发社区创新活力，从而不断孵化新技术。发行版平台和技术孵化器互相促进、互相推动、牵引版本持续演进。

openEuler 覆盖全场景的创新平台



openEuler 已支持 X86、ARM 多处理器架构，持续完善多样性算力生态体验。

openEuler 社区面向场景化的 SIG 不断组建，推动 openEuler 应用边界从最初的服务器场景，逐步拓展到云计算、边缘计算、嵌入式、AI 等更多场景，openEuler 正成为覆盖数字基础设施全场景的操作系统。

openEuler 希望与广大生态伙伴、用户、开发者一起，通过联合创新、社区共建，不断增强场景化能力，最终实现统一操作系统支持多设备，应用一次开发覆盖全场景。

openEuler 开放透明的开源软件供应链管理

开源操作系统的构建过程，也是供应链聚合优化的过程。拥有可靠开源软件供应链，是大规模商用操作系统的基础。 openEuler 从用户场景出发，回溯梳理相应的软件依赖关系，理清所有软件包的上游社区地址、源码和上游对应验证。完成构建验证、分发、实现生命周期管理。开源软件的构建、运行依赖关系、上游社区，三者之前形成闭环且完整透明的软件供应链管理。

2. 平台架构

系统框架

openEuler 是覆盖全场景的创新平台，在引领内核创新，夯实云化基座的基础上，面向计算架构互联总线、存储介质和 AI 发展新趋势，创新分布式、加速引擎和基础服务，结合边缘、嵌入式和 AI 领域竞争力探索，打造全场景协同的面向数字基础设施的开源操作系统。

openEuler 26.09 发布面向服务器、云原生、边缘、嵌入式和 AI 场景的全场景操作系统版本，基于 Linux Kernel 6.6+ 6.18 双内核构建，对外接口遵循 POSIX 标准，具备天然协同基础。同时 openEuler 26.09 版本集成分布式软总线、K3s 边云协同框架等能力，进一步提升数字基础设施协同能力，构建万物互联的基础。

面向未来，社区将持续创新、社区共建、繁荣生态，夯实数字基座。

全场景

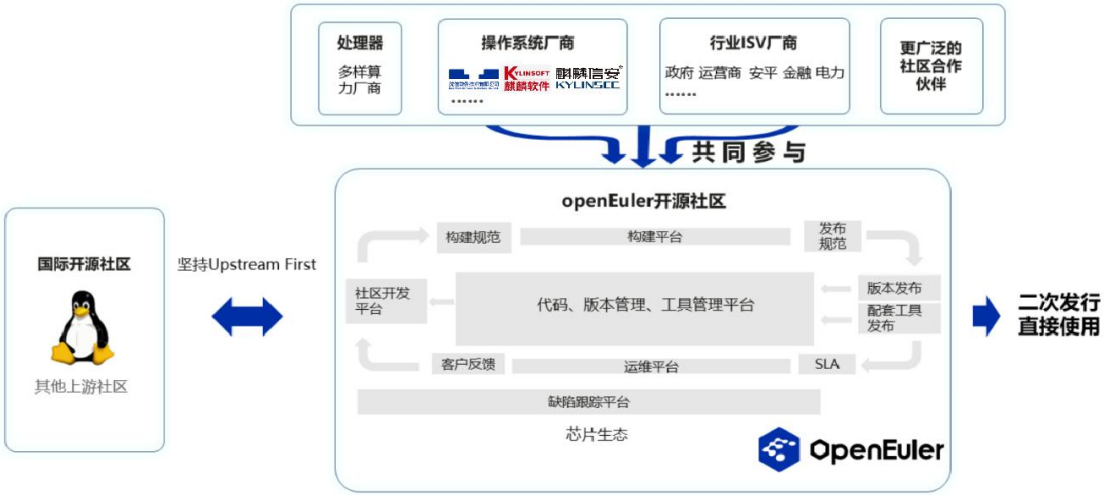
- AI 原生 OS：KV Cache 多级缓存、聚合传输，Agent 语义感知的 KV Cache 调度，支撑推理场景降本增效；Agent 可观测实现零侵入低开销，全链路覆盖，确保 Agent 可靠运行。

繁荣社区生态

- DevStation 开发者工作站：基于 openEuler 的智能 LiveCD 桌面版开发者工作站，旨在提供开箱即用、高效安全的开发环境，打通从部署、编码、编译、构建到发布的全流程。通过集成自研安装工具，可显著压缩社区镜像体积，实现快速下载、安装与试用 ；通过新增 MCP AI 智能引擎，快速完成社区工具链调用，实现从基础设施搭建到应用开发的效率飞跃；通过完善 AI 工具部署环境，可有效提升开发者在 openEuler 环境下集成相关 Agent 应用的开发体验。
- EPKG 软件包：一个绿色轻量的跨平台包管理器，支持 Linux、macOS 和 Windows。创建隔离的 **environments**，安装使用常见发行版（RPM、DEB、Alpine、Arch、Conda、Homebrew、MSYS2）的软件包，无需 root 权限。 每个环境绑定到一个 **channel**（如 Debian、Fedora、Alpine 等指定版本）。注册多个环境，将其二进制文件组合到 PATH 中，在一个 shell 中混合使用不同发行版的软件。

平台框架

openEuler 社区与上下游生态建立连接，构建多样性的社区合作伙伴和协作模式，共同推进版本演进。



硬件支持

openEuler 社区当前已与多个设备厂商建立丰富的南向生态，比如 Intel、AMD 等主流芯片厂商的加入和参与，openEuler 全版本支持 x86、Arm 两种架构，并支持多款 CPU 芯片，包括兆芯开先/开胜系列、Intel Sierra Forest/Granite Rapids、AMD EPYC 4/5 等芯片系列，支持多个硬件厂商发布的多款整机型号、板卡型号，支持网卡、RAID、FC、GPU&AI、DPU、SSD、安全卡七种类型的板卡，具备良好的兼容性。

支持的 CPU 架构如下：

硬件类型	x86_64	arm64
CPU	Intel、AMD、Hygon、兆芯	鲲鹏、飞腾

3. 运行环境

服务器

若需要在物理机环境上安装 openEuler 操作系统，则物理机硬件需要满足以下兼容性和最小硬件要求。

硬件兼容支持请查看 openEuler 兼容性列表：兼容性列表。

部件名称	最小硬件要求

架构	ARM64、x86_64
内存	为了获得更好的体验，建议不小于 4GB
硬盘	为了获得更好的体验，建议不小于 20GB

虚拟机

openEuler 安装时，应注意虚拟机的兼容性问题，当前已测试可以兼容的虚拟机及组件如下所示。

1. 以 openEuler 26.09 为 HostOS，组件版本如下：

- libvirt-9.10.0-37.oe2609
- libvirt-client-9.10.10-37.oe2609
- libvirt-daemon-9.10.10-37.oe2609
- qemu-8.2.0-83.oe2609
- qemu-img-8.2.0-83.oe2609

2. 兼容的虚拟机列表如下：

HostOS	GuestOS(虚拟机)	架构
openEuler 26.09	Centos 6	x86_64
openEuler 26.09	Centos 7	aarch64
openEuler 26.09	Centos 7	x86_64
openEuler 26.09	Centos 8	aarch64
openEuler 26.09	Centos 8	x86_64
openEuler 26.09	Windows Server 2016	x86_64
openEuler 26.09	Windows Server 2019	x86_64

部件名称	最小虚拟化空间要求
架构	ARM64、x86_64
CPU	2 个 CPU
内存	为了获得更好的体验，建议不小于 4GB
硬盘	为了获得更好的体验，建议不小于 20GB

边缘设备

若需要在边缘设备环境上安装 openEuler 操作系统，则边缘设备硬件需要满足以下兼容性和最小硬件要求。

部件名称	最小硬件要求
架构	ARM64、x86_64
内存	为了获得更好的体验，建议不小于 4GB
硬盘	为了获得更好的体验，建议不小于 20GB

嵌入式

若需要在嵌入式环境上安装 openEuler Embedded 操作系统，则嵌入式硬件需要满足以下兼容性和最小硬件要求。

部件名称	最小硬件要求
架构	ARM64、ARM32、x86_64
内存	为了获得更好的体验，建议不小于 512MB
硬盘	为了获得更好的体验，建议不小于 256MB

4. AI 场景创新

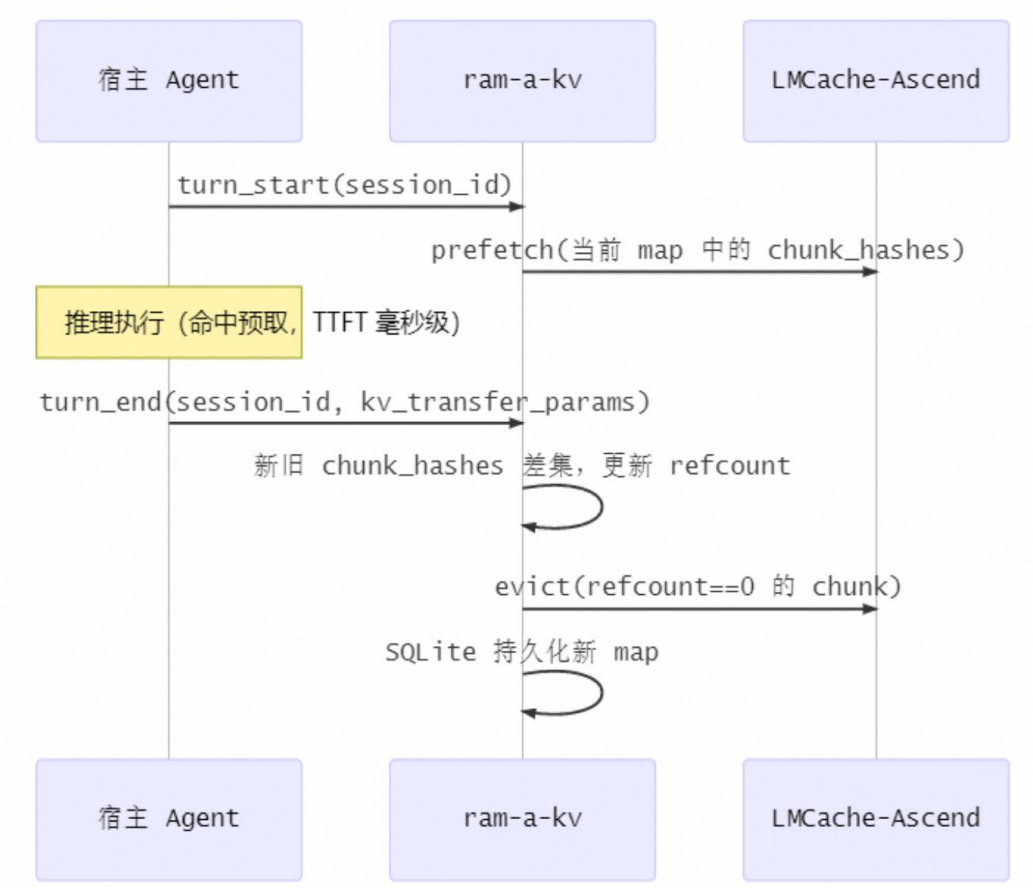
智能时代，操作系统需要面向 AI 不断演进。一方面，在操作系统开发、部署、运维全流程以 AI 加持，让操作系统更智能；另一方面，操作系统作为 AI 的运行环境，需要为 AI 和 Agent 提供异构资源协同管理能力，从而充分发挥 CPU 和 AI 加速器的能力，让业务响应更快，时延更低，当前 openEuler 也率先支持 NVIDIA、昇腾等主流 AI 处理器，成为使能多样性算力的首选。

Agent 语义感知的 KVCache 协同调度

功能描述

在多轮 AI Agent 推理场景中，每一轮推理都依赖此前轮次的中间结果，这些结果以 KV cache 形式驻留在加速卡显存或配套的 KV cache 仓库中。然而现有的 KV cache 系统本质上是被动仓库管理者——仅按 LRU 策略管理缓存，对 Agent 上下文语义无感知。在多租户压力下，当前会话仍需引用的热块常被误驱逐，导致首 Token 延迟（TTFT）从毫秒级飙升到秒级。

ram-a-kv 作为推理栈与 KV cache 仓库之间的语义大脑，将 Agent 上下文映射到 KV cache 块（chunk），预测重用关系，在下一次推理之前主动编排 prefetch / evict，并通过全局引用计数保护跨会话共享的 chunk，最终把 TTFT 压回毫秒级。



ram-a-kv 以独立守护进程形式运行，宿主 Agent 通过 HTTP 事件接入，daemon 据此编排底层 KV cache 后端（LMCache-Ascend）。工作流如上图所示。

核心组件包括：HTTP 事件入口（唯一路由 POST /event，负责鉴权与字段校验）、事件注册表（9 类事件 → handler 分发）、会话状态管理（每会话 chunk map + 全局 refcount，内存互斥锁保护）、SQLite 持久化层（会话 map、turn_count、pins 表）。

事件驱动的生命周期管理

定义 9 个事件覆盖推理轮次完整生命周期：turn_start / turn_end（每轮推理的起止）、snapshot_restore / session_suspend / session_close（会话挂起、恢复与结束）、session_fork / session_fork_end（会话分叉）、session_map（状态查询）、health（健康检查）。宿主 Agent 仅需在关键节点上报事件，daemon 自动完成预取、驱逐与状态更新。

引用计数安全

核心安全不变式：一个 chunk 只有在全局引用计数降为 0 时才会被驱逐，跨会话共享的 chunk 绝不会被某个会话提前删除。turn_end 用新旧 chunk_hashes 做差集更新引用；session_fork / session_fork_end 对称增减引用保护父子会话；daemon 启动时遍历所有 session map 重建 refcount 表，共享 chunk 在重启后仍受保护。

持久化与重启恢复

所有 session map 实时落 SQLite，daemon 重启时遍历全部会话回内存并重建 refcount。session_suspend 保留 SQLite 行便于后续 snapshot_restore 恢复，使会话状态在重启或挂起后无缝延续。

应用场景

场景一、多轮对话 Agent（典型）

问题：用户与 AI 助手多轮对话，每轮推理都依赖前序上下文。随着对话变长，KV cache 总量超过显存，被动 LRU 会把热块驱逐到慢存储，TTFT 飙升。

方案：宿主 Agent 在 turn_start / turn_end 上报事件，ram-a-kv 据此预取本轮所需 chunk、驱逐真正无人引用的块。

收益：TTFT 保持毫秒级，长对话不退化。

场景二、会话挂起与恢复

问题：用户暂停对话（如切换标签页、夜间续聊），KV cache 被回收。次日续聊时若不恢复，需重新计算所有历史 token，开销巨大。

方案：暂停时上报 session_suspend，daemon 落 pin + 释放内存引用但保留 SQLite 行；恢复时上报 snapshot_restore，daemon 从 SQLite 加载 map 重新 prefetch，会话状态延续。

收益：用户感知层面的"无缝续聊"；底层资源按需释放，不长期占用显存。

场景三、会话分叉

问题：从已有会话派生新分支，父会话继续推进时，子会话仍需引用早期共享 chunk。

方案：派生时上报 session_fork，daemon 对当前 chunk refcount +1；子会话独立后上报 session_fork_end，对称地 refcount -1。

收益：分支推理不互相破坏，共享 chunk 在最后一个引用者释放前不会被驱逐。

场景四、多租户推理服务

问题：推理服务承载多个用户/会话，KV cache 后端容量有限，需要在多租户之间公平且高效地调度。

方案：ram-a-kv 的全局 refcount 表天然跨会话；任何 chunk 只要还有引用就不会被驱逐，避免"租户 A 把租户 B 还在用的块挤掉"。

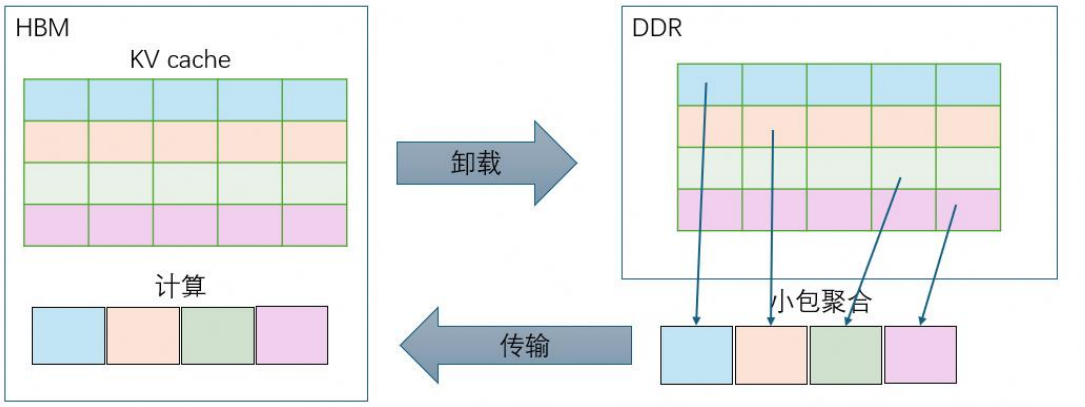
收益：多租户隔离的语义安全；后端容量利用率最大化（无引用即驱逐）。

KV Cache 聚合传输

功能描述

随着大语言模型（LLM）参数规模从百亿跃升至千亿乃至万亿级别，主流推理服务面临的核心矛盾已从"算力不足"转变为"显存不足"，以 DSA 3.2 模型为例，M 级输入序列预计 KV 需要 30G+ HBM 空间，而实际上由于 DSA 3.2 模型的创新改进，实际每次仅需要通过 topK 来参与 attention 计算，这样为 DDR 与 HBM 协同提供了前提。

本特性围绕 sparse attention 方向，实现了 decode 阶段 kv cache 在节点内 DDR+HBM 协同，在 prefill 结束后，将 full kv 卸载到 DDR，HBM 仅放 indexer kv 以及为了保持一定命中率的部分 KV，在实际 layer attention 阶段按需加载，采用 MemFabric 组件聚合传输数据，有效缓解在高并发场景，由于 HBM 内存不足，导致部分请求阻塞的情况，有效缓解高并发场景降低时延，提升 BS 及 TPOT。



主要功能点，基于 vllm 版本优化：

1、H2D 传输优化，有效提升原生有效带宽问题。

2、优化 kv 缓存命中率，保持命中率 90%以上，减少单次数据传输。

应用场景

在 LLM 推理加速 PD 融合/PD 分离场景，实现对 decode 阶段优化，在相同的硬件配置下，充分利用 host 侧内存作为 HBM 的扩展内存，基于 vllm offload 能力优化，降低高并发场景部分请求等待，提升 BS 及 TPS 。

KV Cache 多级缓存

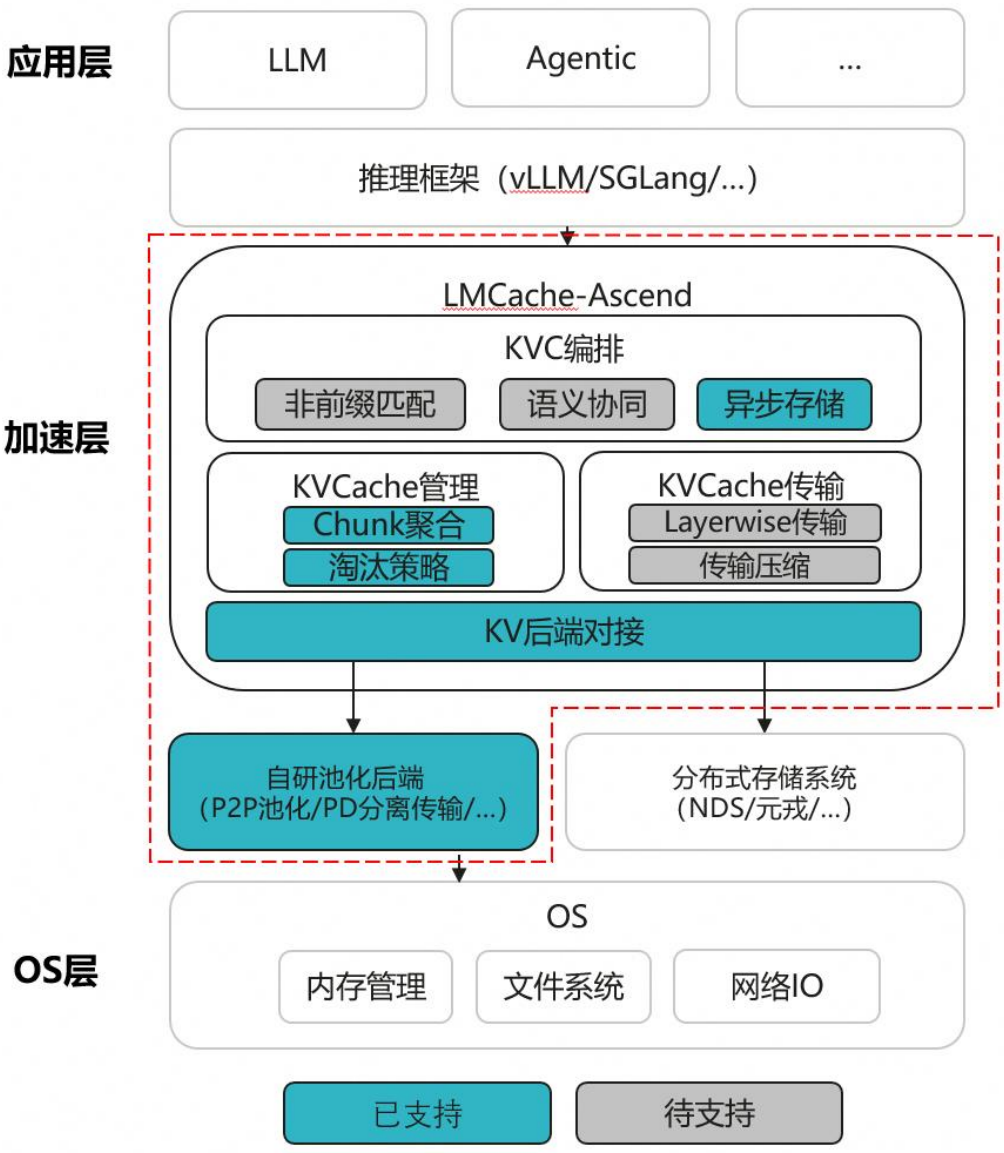
功能描述

从单轮问答到多轮对话，再到如今的 Agent 工作流，大模型的交互模式已经截然不同。一个典型的 Agent 任务（如代码生成、科研分析、自动化办公）往往包含数十次甚至上百次的工具调用、思考回溯和上下文延续，这也给 KVCache 管理后端带来了挑战：

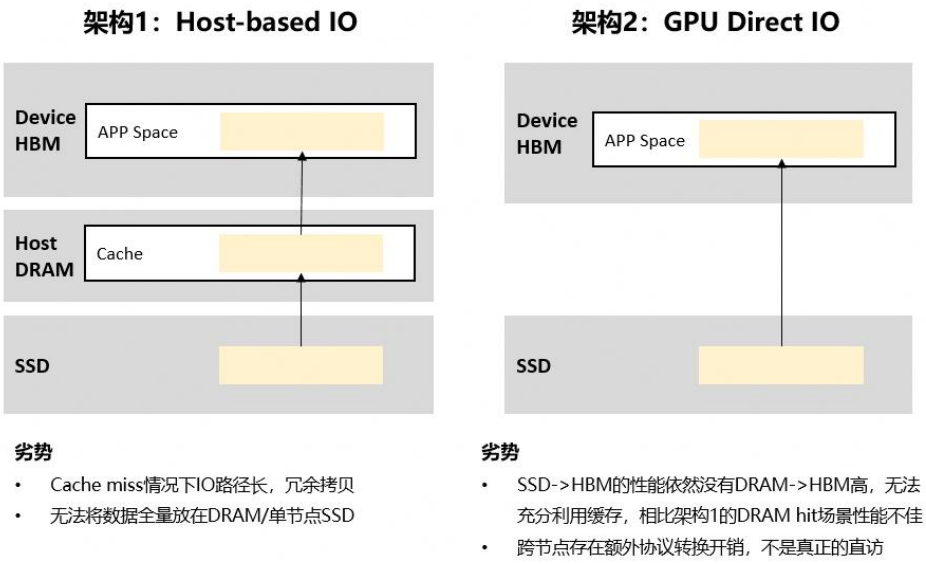
单请求 KVCache 量级质变：Agent 要维持完整对话历史、检索片段和中间推理，上下文动辄 128K-1M，单请求 KVCache 从 MB 级飙到 GB 级，HBM 放不下，必须分级卸载到 DDR/SSD/超节点池化——后端不再只是“搬运工”，而是要管理一个跨越多级存储的 KVCache 生命周期，搬运调度本身成为瓶颈。

前缀可复用性急剧上升：多 Agent 实例共享相同的 system prompt、角色设定、知识库前缀，KVCache 命中率极高——后端需要能够毫秒级定位最长前缀命中的缓存块，以存代算跳过重复 prefill。

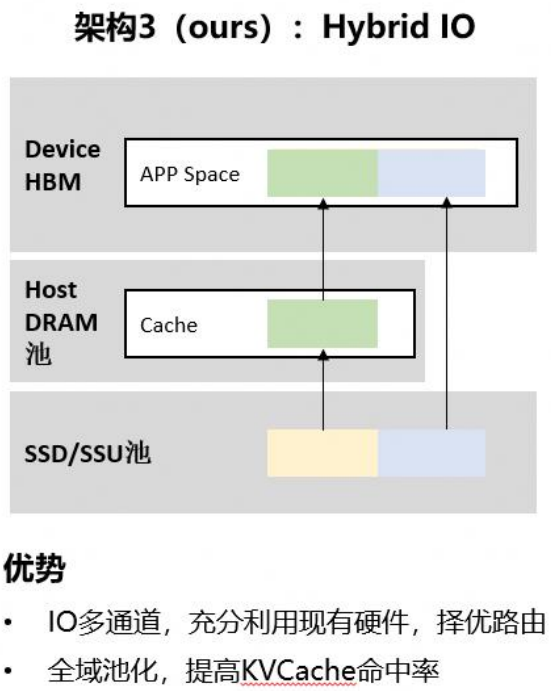
为了应对上述挑战，构筑异构融合多级 KV 缓存池 LMCache-Ascend 不仅增强了自研 KVC 池化及 PD 传输能力，同时提供 storagebackend 生态接口，兼容 mooncake、元戎 datasystem 等高性能分布式存储后端，端到端性能再度提升。



随着 LLM 模型规模变大，HBM 和 DDR 已经无法容纳每日请求产生的 KV Cache，并需要卸载到 SSD，比如 Deepseek 集群每天生成 776B token，KV Cache 的数据量大致为 14.5PB。但是当前主流的 SSD 卸载方案在性能存在上存在不足。



为了提高 KV Cache 的整体性能，实现混合式 IO，不仅通过直通发挥存储介质的带宽，也通过缓存充分利用 Host DDR 的带宽。还通过池化技术进一步聚合读写带宽。



一、LMCache 多级缓存

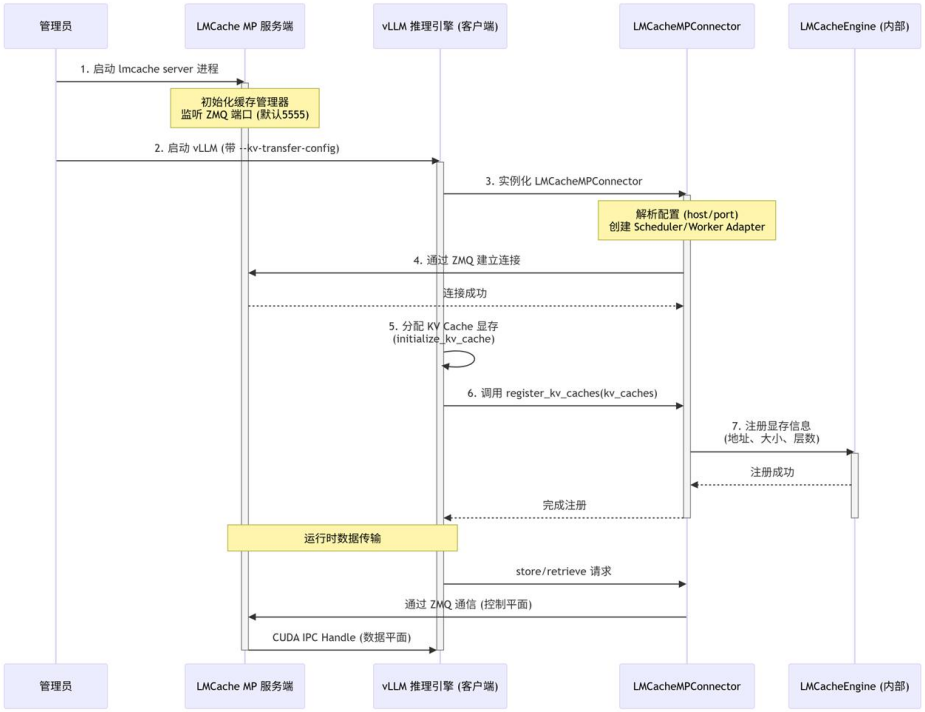
LMCache 的整体技术架构围绕一个核心目标构建：将 KV Cache 从推理引擎的进程内，提升为一个可跨进程、跨节点共享的独立缓存层。它通过“连接器层 + 引擎层 + 存储层”的分层设计，以及“进程内组件 + 独立服务进程”的混合部署模式来实现这一目标。

层次	核心职责	关键组件
连接器层（Connector）	作为 vLLM 的插件，拦截 KV Cache 操作，在 vLLM 的物理块视图与 LMCache 的逻辑 Token 视图之间进行转换与调度	LMCacheConnector、RequestTracker、ReqMeta
引擎层（Engine）	核心控制器，协调缓存生命周期、I/O 操作、分层存储策略执行	LMCacheEngine、TokenDatabase、StorageManager
存储层（Storage）	管理多级存储后端（内存、P2P、磁盘、远程），执行数据写入与检索策略	LocalCPUBackend、P2PBackend、LocalDiskBackend、RemoteBackend

LMCache 接入 vLLM/SGL 推理引擎：

服务端就绪与监听：LMCache 服务端进程启动后，会初始化内部的缓存管理器，并在指定的 ZMQ 端口上开始监听来自客户端的连接请求。

```
vllm serve <model> \
    --kv-transfer-config \
    '{"kv_connector": "LMCacheMPConnector", "kv_role": "kv_both"}'
```



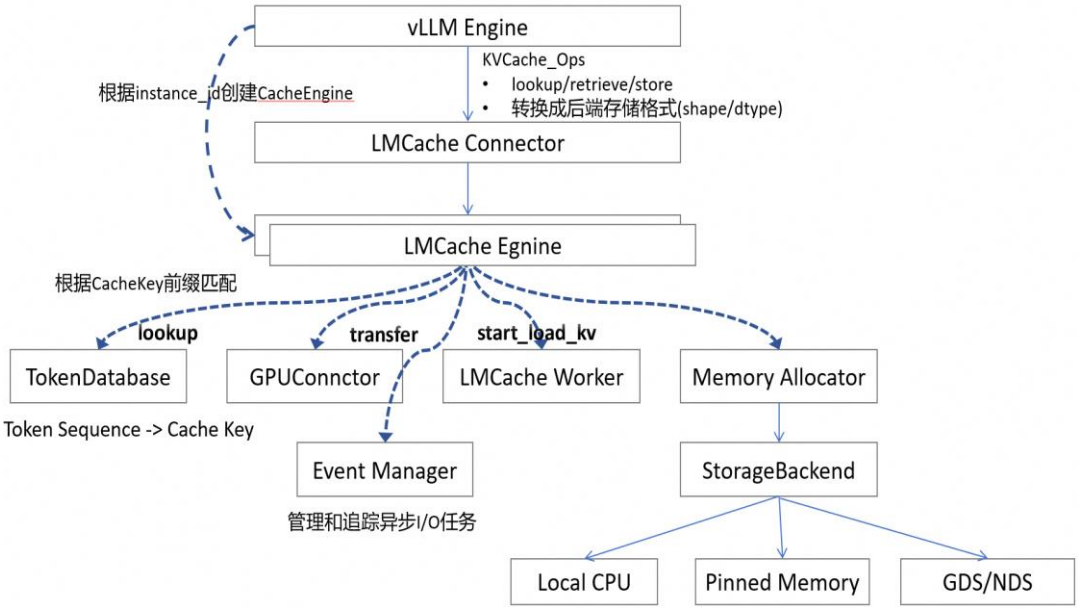
vLLM 连接与注册：vLLM 进程在初始化时，会根据 --kv-transfer-config 中的配置，实例化 LMCacheMPConnector。这个连接器会通过 ZMQ 协议，向配置的 LMCache 服务端地址发起连接。连接成功后，vLLM 会调用 register_kv_caches 接口，将本进程内分配的各层 KV Cache 显存信息（包括内存地址、大小、层数等）注册到 LMCache 服务端。

运行时数据传输：初始化完成后，vLLM 便可通过 LMCacheMPConnector 与服务端进行数据传输。

存储（Store）：vLLM 生成 KV Cache 后，连接器会先执行 Gathering 操作，将分散在多个显存页（Paged Memory）中的 KV 块聚合到一个连续的缓冲区中。

高效传输：为了高效地跨越进程边界，MP 模式优先使用 IPC（Inter-Process Communication） 机制。工作进程不会拷贝整个数据，而是将显存导出为轻量级的 IPC Handle 发送给 LMCache 服务端。服务端导入该句柄后，便能直接访问工作进程的 GPU 内存，实现几乎零拷贝的高效数据传输。

整体技术组件：



LMCache 通过 KV Connector 对接推理引擎接口：

操作阶段	接口名称	核心职责
初始化与注册	register_kv_caches	在 Worker 初始化时，将 vLLM 分配的 KV Cache 显存张量注册给 Connector，便于后续管理
查找与决策	get_num_new_matched_tokens	调度器在调度请求时调用，查询除了本地已计算的 Token 外，还有多少新的 Token 可以从外部 KV Cache 中加载
预加载与状态同步	update_state_after_alloc	在 vLLM 为请求分配好物理块后调用，将物理块信息和需要加载的外部 Token 数量同步给 Connector，为后续的加载做准备
启动异步加载	start_load_kv	Worker 侧在模型前向传播开始前调用，启动一个异步任务去加载外部 KV Cache

逐层等待加载	wait_for_layer_load	在模型的每一层注意力计算前调用，会阻塞直到该层所需的 KV Cache 数据已就绪，实现流水线加载
逐层异步保存	save_kv_layer	在模型每一层计算完成后调用，将该层生成的 KV Cache 异步保存到外部存储
等待保存完成	wait_for_save	在模型前向传播结束时调用，阻塞直到所有层的异步保存操作都完成，确保数据安全
生命周期收尾	request_finished	在请求处理完成、准备释放其 KV Cache 块时调用。若返回 True，表示 Connector 将异步接管块的生命周期管理
查询异步完成状态	get_finished	调度器通过此接口查询哪些请求的异步保存/加载操作已经完成，以便安全地释放相关资源

二、NDS 直通

当前 NPU 的文件读写都是通过 CPU 中转，CPU 中转的 2 个问题是：

(1) 额外的数据复制（SSD -> DDR -> HBM）

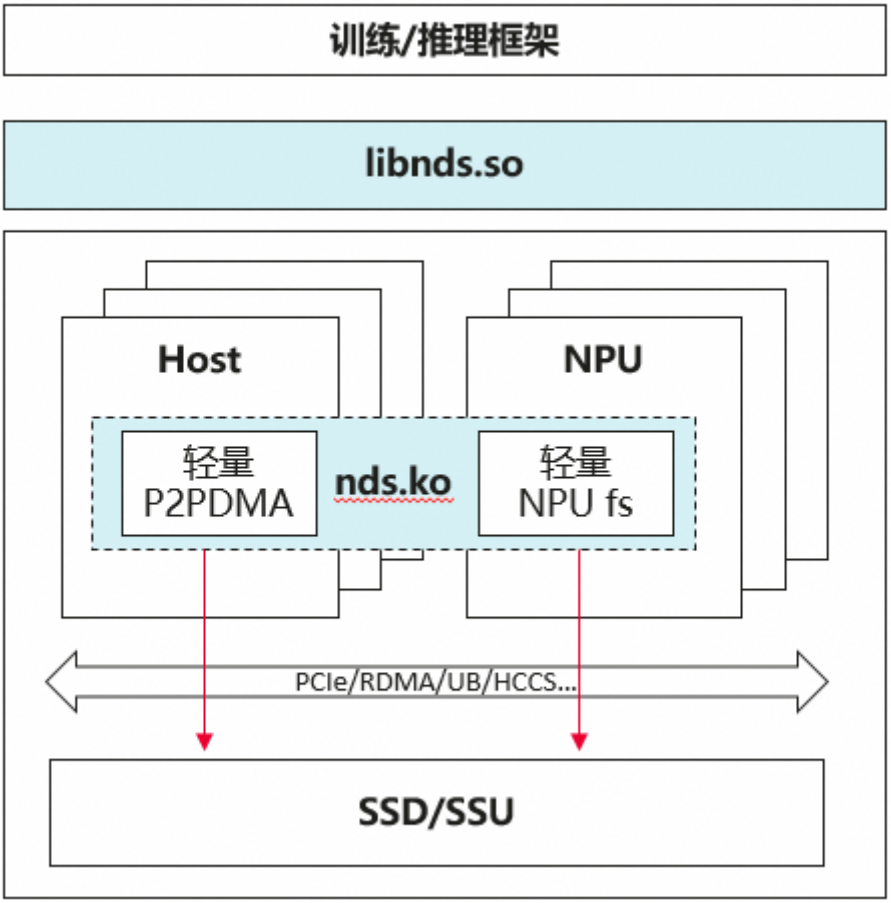
(2) CPU 与 NPU+存储之间的总线带宽 可能小于 NPU 和存储的总线带宽

可以通过数据面直通解决这 2 个关键问题，达到提高性能和减少 Host CPU 占用和 DDR 带宽的目标。NDS 有如下的设计目标：

(1) 免侵入式修改：内核存储栈和驱动免侵入式修改

(2) 低开销：降低/卸载控制面开销、降低内存开销

(3) 高并发高性能：带宽超越 GDS 10%



NDS 是一个端到端框架，能够实现 NVMe SSD 与 NPU HBM 之间的直接数据移动，消除了不必要的主机内存参与。包括了：

1. 内核模块

无需侵入式内核修改，没有 GDS 的 Shadow Buffer 内存开销，实现数据面 bypass CPU。通过实现字符设备，整合文件系统、块层和 NVMe 驱动，直接构造 NVMe SGL 命令，支持发起和查询 IO 操作。

2. 接入 LMCache 生态

实现了一个 LMCache 池化存储后端，将 NDS 集成到 LMCache，透明替换传统磁盘加载路径，包含了池化的 KVCache 文件管理、LRU、Promote 等功能。

NDS 接口提供 C 和 python 两种接口，以 C 接口为例：

接口分类	C 接口
初始化	int nds_init(struct nds_init_param *param) int nds_exit(void)
文件系统注册	int nds_register_fs(const struct nds_fs_desc *desc) int nds_unregister_fs(const struct nds_fs_desc *desc)
HBM 注册	int nds_register_mem(void *addr, uint64_t size, int flags) int nds_unregister_mem(void *addr, uint64_t size, int flags)
HBM 读写：管理 IO 上下文	int nds_io_new_ctx(const struct nds_io_ctx_param *param, struct nds_io_ctx **ctx) int nds_io_destroy_ctx(struct nds_io_ctx *ctx)
HBM 读写：下发读写请求	int nds_io_submit(struct nds_io_ctx *ctx, int nr, const struct nds_io_cb *iocb)
HBM 读写：等待读写请求完成	int nds_io_getevents(struct nds_io_ctx *ctx, int min_nr, int nr, struct nds_io_event *events, struct timespec *timeout)

应用场景

LMCache 池化系统的核心价值，在于将 KV Cache 从“推理引擎内部的临时数据”转变为“可跨请求、跨实例复用的持久化资源”。它的应用场景主要围绕 “减少重复计算” 这一目标展开，以下是几个最典型的场景：

多轮对话与长会话

在多轮对话中，每一轮都会带上之前的历史记录，这些历史记录的 KV Cache 是完全相同的。如果没有 LMCache，每次请求都需要把整个历史重新 Prefill 一遍，造成巨大的算力浪费。LMCache 可以将前几轮对话产生的 KV Cache 缓存下来，后续轮次直接复用。实测数据显示，在多轮对话场景下，LMCache 能将首 Token 延迟（TTFT）降低 4.4 倍，同时支撑的活跃用户数从 2 个提升到 8 个；对于长会话场景，实测 TTFT 加速比可达 37 倍。

P/D 分离架构中的 KV 传输

在 Prefill-Decode 分离部署中，Prefill 节点负责处理输入，Decode 节点负责生成。Prefill 节点计算完的 KV Cache 需要高效地传递给 Decode 节点。

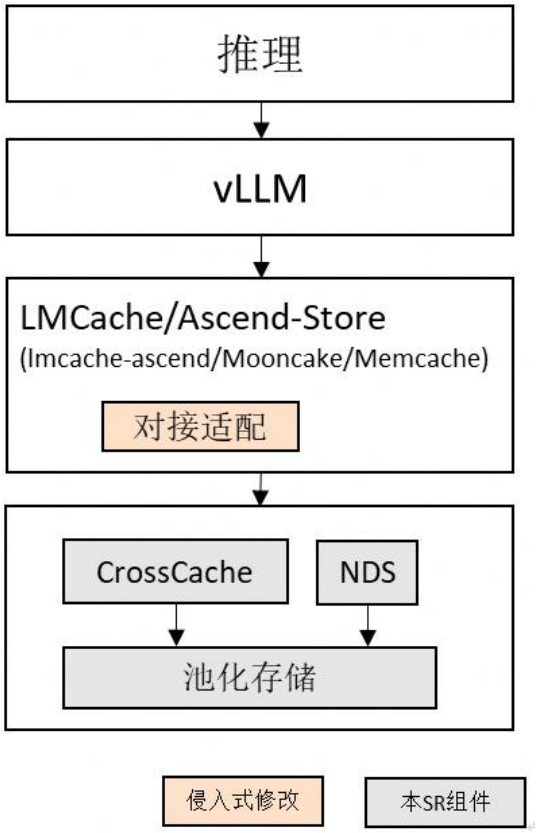
LMCache 在这里扮演实时传输层的角色。它通过 Gather/Scatter 机制，将分散在 vLLM Paged Memory 中的 KV 块聚合成大 Chunk，再通过 NIXL 或 Mooncake 的 RDMA 通道推送到 Decode 节点。同时，LMCache 的 MultiConnector 能力允许一个 Prefill 节点同时进行 P/D 传输（给 Decode 节点）和本地 Offload（给共享缓存），实现“一次计算，多处使用”。

跨实例共享与负载均衡

在标准的 vLLM 部署中，多个推理实例通常各自维护独立的 KV Cache，形成“缓存孤岛”。同一个请求如果落到不同的实例上，即使内容完全相同，也无法命中对方的缓存。

LMCache 的 P2P 模式和 MP（多进程）模式解决了这个问题。多个 vLLM 实例可以通过 LMCache 共享同一个 CPU 内存池中的 KV Cache，或者通过 NIXL/RDMA 从对等节点直接拉取。

当前 NDS 直通的主要场景用于推理服务的 KV Cache 加载加速，整体视图如下所示，数据面主要涉及 LMCache/LMCache-Ascend（Ascend-Store 暂未对接）和 NDS 的对接，对接需要在 LMCache 和 LMCache-Ascend 中通过侵入式修改实现，不涉及其它第三方组件的修改。



已完成收益及路标：

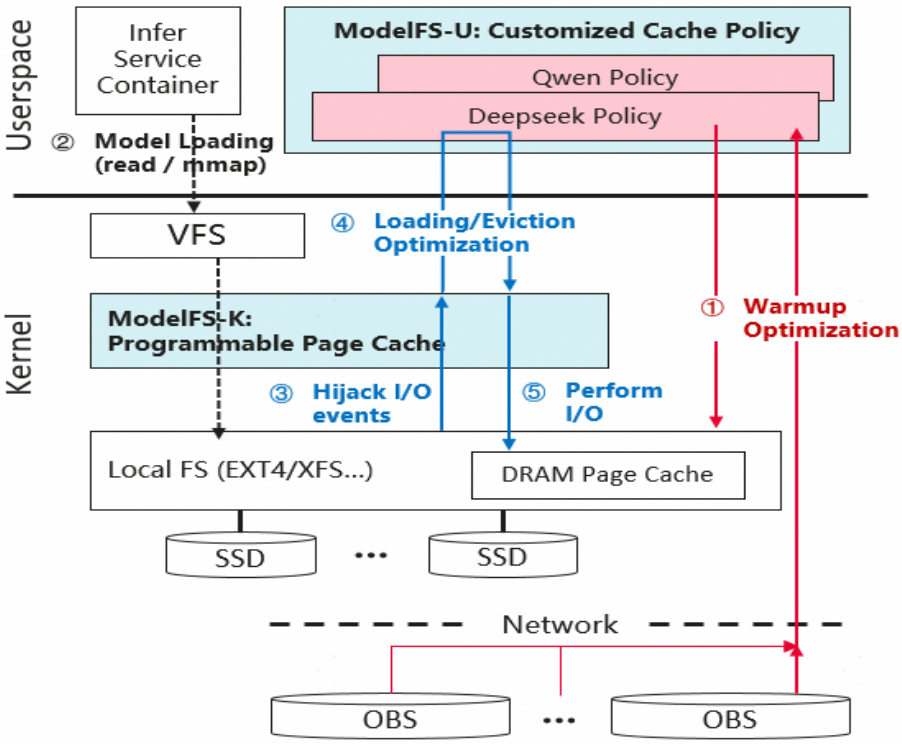
服务上线：Minimax 2.5/2.7，GLM5，Qwen 3.5 均上线 Coding 服务，TTFT 降低 25%。

当前完成 UBSIO 和 NDS 的对接。

ModelFS 用户态模型加载优化

功能描述

ModelFS 针对大语言模型（LLM）推理启动阶段的模型加载瓶颈进行优化。 现有相关工作主要通过优化推理框架来提升模型加载性能，这种方法往往以牺牲兼容性为代价。然而，基于我们的工业实践经验，兼容性是决定一项技术能否在实际场景中广泛应用的关键因素。 本工作在保证强兼容性的前提下，通过优化文件系统的缓存策略，实现了模型加载性能的领先水平。 ModelFS 在内核中设计了一个非侵入式、灵活且轻量级的可编程页缓存框架，允许用户自定义文件系统的页缓存策略。 基于可编程页缓存，我们进一步设计了面向模型加载优化的缓存策略的参考实现。



ModelFS-K：

ModelFS 的内核模块，提供文件系统（FS）页缓存可编程框架。核心设计是一个堆叠 FS，可以挂载到现有 FS 之上，ModelFS-K 将底层文件系统的相关逻辑用 UPC（用户态调用）重定向到用户态实现的 prefetch 和 evict 函数。

ModelFS-U：

ModelFS 的用户态模块，提供缓存策略的运行时。提供一套 VFS-like 的用户态编程框架，模型实现者/IO 优化者可以根据模型的加载 IO 特性自定义缓存策略。用户需实现 init()，exit()， prefetch()，evict()，ModelFS-U 会将它们注册到 ModelFS-K 中，并在运行过程中负责解析 IO 事件和进行异步数据预取/淘汰。

应用场景

ModelFS 可用于多种类型存储后端的 IO 加速，包括本地 FS，分布式 FS，对象存储（OBS）等，以下是两种典型的用法。

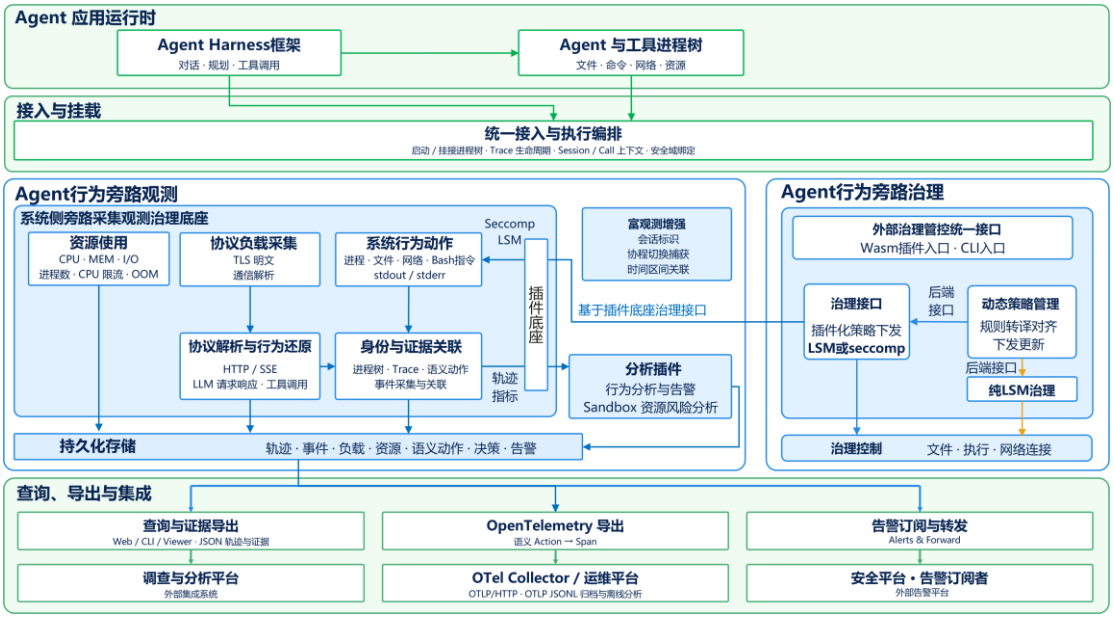
1. 远端 OBS 存储权重预热：对于模型权重存储于远端 OBS 的情况，由于 OBS 带宽低，推理系统往往会将热点权重提前预热到本地存储。ModelFS 可以通过实现 IO 聚合的缓存策略，将多个 OBS 桶的带宽聚合来加速权重预热。

2. 本地 SSD 权重加载加速：对于模型权重存储在本地 SSD 的情况，性能瓶颈在于本地 SSD 带宽利用率低。ModelFS 实现了一种基于 I/O 模板的机制，在第一次运行时追踪每个推理服务的 IO 加载行为，而在后续推理启动过程中可以精确感知未来的 IO 行为，进而充分利用 SSD 带宽、XPU 亲和性以提升预取与淘汰机制的效率。

Agent 可观测和治理

Agent 已经不再局限于单轮问答或代码补全，而是逐步演进为具备多轮推理、工具调用、文件读写、脚本执行、外部系统访问和跨 Agent 协作能力的复杂系统。换言之，相比传统的工作流方式，当前 Agent 具备了更高的自由度，随之带来 Agent 行为的不确定性及安全可靠性风险。而且 Agent 逐步进入生产环境，企业关心的问题也随之变化。过去，一个系统是否稳定，主要看服务是否在线、接口是否超时、错误码是否异常；现在，一个 Agent 是否可信，还要看它在完成任务的过程中到底做了什么、为什么这么做、造成了哪些影响，以及事后能不能全链路审计，是否可以进行异常行为的治理控制。

openEuler 提供了一整套面向 Agent 可观测及运行治理的能力，包括基于系统旁路观测和治理的服务和插件框架底座 AcTrail，及扩展富观测并提供安全治理防护的 AgentCensor。



功能描述

1、旁路观测 AcTrail

AcTrail 是一套面向 Agent 的旁路观测和治理的服务和插件框架底座，通过无侵入的系统侧观测与跨层行为关联，将系统行为、模型交互和敏感访问处置统一到可追溯的 Agent 语义行为轨迹中，并提供灵活控制治理的底座能力，实现 Agent 的运行安全、稳定可靠、持续优化。

具体地，针对跨多框架的 Agent 运行时行为，它能够基于自动化的挂载点探测，非侵入地基于用户态和内核态探针，采集 Agent 运行用户态协议负载（如 TLS 接口访问）、内核态事件（进程、文件、网络、Bash 指令等）、资源访问情况（如 CPU、OOM 等），实现 Agent 的关键系统行为观测全覆盖、防绕过。对于传统可观测无法追踪的工具动作声明外的衍生影响也能够进行关联结合，从而识别 Agent 工具声明以下的动作行为。此外，对于传统可观测所无法识别的 Agent Harness 框架异常，也能够完整观测其异常的探测、上传行为。

同时，它基于底层观测事件（如请求片段、fork 调用、读写操作等）组装还原上层动作运行视图，将关联零散事件为具体 Agent 动作，从系统侧还原整个 Agent 行为视图，将的 LLM 响应、Agent Harness 行为、Agent 工具调用、系统环境动作影响进行关联构成完整细粒度的 Agent 行为影响轨迹，同时提供预埋、跨框架、可编程的控制位点，支撑 Agent 安全分析控制、可靠性增强和性能优化。

2、富观测 AgentCensor

智能体的一次任务会跨越多轮交互和多个工具调用，产生分散的执行与资源访问记录。单独的事件或日志难以说明哪些操作属于同一会话、操作之间存在什么联系，以及异常发生在哪个调用环节，因此需要在事件记录之上建立统一的执行上下文。

富观测能力以会话级事件关联为核心，基于运行框架提供并可传递的会话标识，将模型交互、工具执行和资源访问记录归入同一上下文。围绕会话组织调用、操作对象和执行结果，形成从应用行为到系统操作的关联视图，呈现任务执行过程及其涉及的资源访问。

关联机制通过会话标识确定事件归属，结合调用标识、进程关系和事件时间建立跨层联系。对于框架提供调用起止信息且能够唯一匹配的事件，进一步辅助关联到具体调用。事件记录、关联关系及证据引用统一保存在 SQLite 数据库中，并建立会话与时间、调用与时间的联合索引，支持从应用调用追溯相关系统行为及其关联依据。

在此基础上，通过数据库只读视图支持按会话查询事件序列和关联行为，追踪已关联调用涉及的命令、文件和网络访问，并从异常事件反向查找所属会话及调用上下文。关联结果支持 JSON 导出，可用于执行过程分析、异常定位和安全审计，减少跨日志查找与人工拼接的工作。

3、动态控制治理

智能体的执行路径和资源需求往往无法在任务开始前完全确定，工具调用还可能触发文件修改、程序执行或外部连接。过宽的固定权限会扩大风险，过严的限制又会影响正常任务，因此需要在执行过程中落实安全约束，并支持根据任务需求调整访问策略。

动态控制治理能力面向具备 eBPF LSM 支持的内核环境，将安全策略落实到资源访问路径。在文件打开与权限检查、程序加载、网络连接等 Linux 安全模块（LSM）钩子上挂接 eBPF 程序，由内核同步查询规则并作出允许或拒绝判定，对违规操作返回权限错误，实现运行时安全拦截。

控制层将目标进程树绑定到治理域，通过 BPF map 保存进程、治理域与策略之间的映射关系，以及文件、程序和网络访问规则。子进程创建时继承所属治理域，后续操作按当前域策略检查，约束可访问的文件范围、可执行的程序以及连接目标地址和端口。

策略更新采用双规则区机制：先校验新策略并写入备用规则区，再切换生效规则区，无需重启受治理进程。新策略对后续触发安全检查的操作生效，并支持按需回滚至历史策略。同时按配置记录进程、治理域、判定结果及策略版本，为拦截原因核查和策略追溯提供依据。同时，对用户提供基于底层观测事实和还原动作的分析、控制的可编程接口，支持用户可编程式接入插件控制插件以实现更加丰富和复杂的控制逻辑，供用户按需保障 Agent 的运行安全。

应用场景

1. 可回放、细粒度证据链与事后审计

将模型返回 → 工具调用 → 系统调用（进程拉起、文件访问、网络访问、指令执行）组装成完整的调用链和调用动作树。事后复盘可直接回放，无需拼凑日志，满足合规审计与事故追溯需求。

2. 人机协作的复核点

当判定需要人工介入时，事务挂起并等待人工决策：批准则继续执行，拒绝则自动回滚。由此避免 Agent 自行耗到超时，或做出不可逆操作。

3. 防御 Agent 失控与供应链攻击

针对 Agent 执行破坏性命令（如删库）、被注入恶意输入（如密钥窃取）、安装被投毒依赖等场景，通过执行前的同步判定与写入隔离，将破坏控制在可回滚范围内，弥补“提示词 + 沙箱”防线在 Agent 内部工具调用层面的缺口。

数据引用减少上下文

功能描述

Agent 执行长任务时，消息历史中会不断累积大体积的工具输出（读文件结果、命令执行输出、网页抓取内容等），这些内容在后续轮次中大部分不再被使用，却持续占用上下文窗口并推高 token 成本。SCCS 的处理方式是：

压缩替换（卸载原文）：对成功返回且体积较大（>3000 字符）的工具输出，将其原文从上下文中卸载，替换为一条轻量的引用表示——`[REF_ID: xxx] (Summary: ...)`，即一个引用标识加一段结构化摘要（包含代码行号、函数签名等关键结构信息），原文暂存于存储后端；

按需取回（恢复原文）：当模型在后续任务推进中确实需要某段被压缩的原文时，调用 `fetch_original_data(ref_ids=[...])`，SCCS 按 ref_id 返回完整原文，实现精确到引用粒度的内容恢复。

整个流程构成"**卸载 → 摘要 → 按需恢复**"的闭环，等价于给 Agent 的上下文装了一个"外部内存 + 换页机制"：上下文中常驻的是索引（摘要），数据本体按需换入。

按需加载生效的前提是模型知晓引用机制的存在与用法。SCCS 通过三条链路同时保障：

链路	内容	作用
工具定义	`fetch_original_data` 工具（名称/描述/参数）注入模型工具集	让模型知道"可以取回"

压缩内嵌提示	每个被压缩的结果自带 `[SYSTEM NOTE - REF_ID HANDLING]` 系统提示	在压缩发生点即时告知"如何取回"
回合适时上下文注入	每轮向 prompt 追加 `REF_ID_INSTRUCTION` 决策指南（最高优先级决策指令）	持续指导模型"何时取、怎么取"

压缩触发策略：并非所有工具输出都值得压缩，SCCS 的触发条件为同时满足：执行成功（报错结果不压缩，直接透传，避免模型丢失最需要的错误信息）、有文本内容、体积超过阈值（>3000 字符）、非取回结果（防止取回-再压缩循环）。压缩热点集中在读大文件（read）、大输出命令（exec，如 grep/git log）与网页抓取（web_fetch）三类场景。

零侵入接入：以宿主 Agent 框架的原生插件机制接入（如 OpenClaw 的 `agent --local` 激活 + `tools.alsoAllow` 交付取回工具），无需修改宿主代码；
会话/进程隔离：每个任务会话使用独立状态目录与 session-key，取回互不串扰；
可插拔存储后端：支持进程内存（`memory`，不落盘）等后端，原文生命周期与部署形态可按需选择。

应用场景

长会话、多工具调用的 Agent 任务：延长 Agent 在有限上下文窗口内可支撑的任务深度。

上下文窗口受限的部署环境：降低输入侧开销，容纳更长的任务历史或更复杂的任务编排。

自动化代码修复/软件工程任务：验证了“摘要足够定位问题、原文需要时可取回”的工作模式。

批量评测与生产化 Agent 服务：控制上下文规模，为规模化、可复现的评测与服务提供稳定保障。

RAG/多源信息消费型应用：适用于客服问答、报告生成、深度调研等“读多写少”的场景。

SkillHub

功能描述

SkillHub 是面向 AI Agent 生态的 Skills 检索与分发平台，由 openEuler 社区建设运营。平台把分散在各类代码托管平台上的 AI 技能统一汇聚、自动评估并公开分发，让开发者能够放心地发现、评估、使用和贡献 AI 能力。

平台主要提供以下功能：

- 技能发现：以自然语言或关键词描述需求即可检索到匹配的 AI 技能，并可按领域、贡献者来源、安全等级等维度进一步筛选，快速定位所需能力。
- 安全评估与信息透明：每个上架技能都经过自动化的安全评估，并公开呈现量化的风险评分与直观的安全等级，让使用者在选用前就能清楚了解其风险状况。平台提供的是客观的风险信息，供使用者结合自身场景自行判断与决策。
- 版本追溯：技能的历史版本被完整保留，用户可以查看版本演进、选择需要的版本下载使用。
- 便捷获取：找到合适的技能后，可通过一条命令安装到本地 Agent 环境，也可下载技能包自行集成。
- 开放贡献：贡献者只需提供一个包含技能说明文件的公开仓库，并在社区贡献仓库中登记即可完成上架，审核通过后由平台自动完成后续工作。
- 生态汇聚：平台持续自动收集社区与三方技能，形成不断增长的技能库，并通过使用热度反映各技能的受关注程度。



应用场景

Agent 开发者发现并接入能力：开发者描述所需能力即可检索到匹配技能，查看安全等级与使用说明后一键安装，无需再逐个仓库翻找和手工集成。

企业引入第三方技能前的风险评估：企业可按安全等级筛选技能，查看公开的风险信息与评估结论，结合自身业务场景判断某项能力是否适合引入，把风险识别前置到选取阶段。

社区与企业对外发布沉淀能力：社区 SIG、企业组织和个人开发者只需提交一次登记，即可将内部沉淀的可复用能力对外发布，后续更新由平台自动跟进，无需承担平台运维工作。

第三方工具消费技能生态：IDE 插件、内部工具市场等可接入平台能力，把技能目录集成到自身产品中，让用户在熟悉的工具里直接获取 AI 能力。

生态冷启动与持续增长：平台自动汇聚既有社区技能形成初始内容，再通过低门槛的社区贡献与使用热度反馈，推动技能生态持续增长。

5. 服务器场景创新

oeAware 场景化智能调优 Skill

功能描述

oeAware 场景化智能调优 skill 是面向操作系统层性能调优的智能化能力集合，由 witty-opentunex（编排元技能）、opentunex-data-collection（数据采集）、opentunex-scenario-bottleneck（场景化瓶颈分析）、opentunex-scenario-tuning（场景化调优执行）四个核心 skill 协同构成，将依赖专家经验的瓶颈定位与参数调优过程转化为由大模型推理驱动的、可远程执行、可灰度回滚的自动化调优流水线。

模型选择：建议使用 GLM-4.7 或 MiniMax-M2.7 及以上能力的模型（上下文长度 > 200K）。

模块采用四层结构：元技能编排层负责流程编排与模式判定；三阶段流程层对应数据采集、瓶颈分析、调优执行；场景化子能力层共部署 9 个场景化分析子技能与 9 个场景化调优子技能；文件契约产物层以统一路径约定串联上下游。

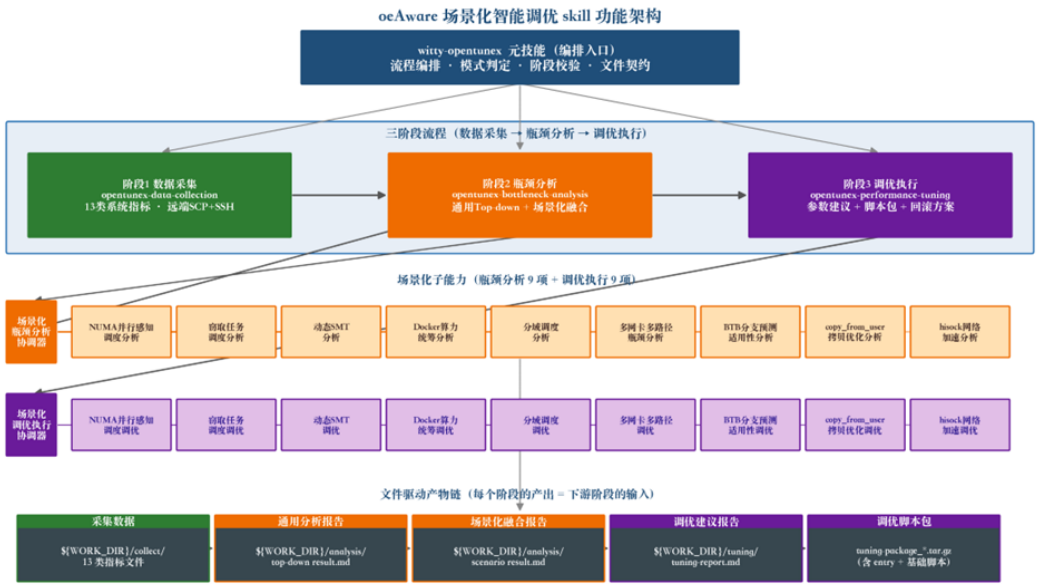


图 1-1 oeAware 场景化智能调优 skill 功能架构图

应用场景

oeAware 场景化智能调优 skill 适用于数据库、缓存、消息队列、大数据等业务在操作系统层遇到的性能调优场景。用户通过 agent 集成该 skill 工具包，调用 witty-opentunex 发起会话，由 agent 自动完成数据采集、OS 层瓶颈分析与场景化调优方案生成，达到快速定位瓶颈并落地可回滚调优脚本的效果。

场景与业务负载映射总览

场景	典型业务负载	可识别的瓶颈现象	对应调优能力
NUMA 并行感知调度	多 NUMA 服务器上的内存数据库、HPC、大模型推理	跨 NUMA 访问率高、内存带宽受限、远端内存命中率高	PARAL（aarch64）
窃取任务调度	高 CPU 利用率的多核通用业务	CPU 负载严重不均衡、%steal/sys 偏高、空闲核无法被利用	STEAL（aarch64）
动态 SMT	低 CPU 利用率且存在超线程干扰的业务	CPU 利用率低、单线程延迟受超线程邻居干扰	KEEP_ON_CORE
Docker 算力统筹（burst）	容器化部署的数据库/微服务/在线业务	容器 CPU 限流丢任务、宿主机仍有空闲核	sched_soft_runtime_ratio + cpu.soft_quota
分域调度	多 NUMA + 小配额多实例容器/进程场景	跨 NUMA 漫游抖动、调度域粒度过粗	SOFT_DOMAIN（aarch64）
网卡多路径	多网卡多 NUMA 的网络密集型业务	跨 NUMA 中断亲和差、网络抖动	oenetcls/venetcls + ntuple
BTB 分支预测	鲲鹏 920 + redis/mysql 等线程数多、关键线程切换频繁的业务	分支预测记录相互污染、关键线程 IPC 下降	BIOS 禁用 TidCMP
copy_from_user 拷贝优化	ARM64 Hisilicon CPU 上有大块读写热点的业务	perf 看到 copy_from_user/_to_user 占比较高	ldp/ldtp 内核补丁
hisock 网络加速	短连接/高 PPS 的网关、代理、NFV 业务	nf_hook 热点明显、L2/L3 netfilter 开销大	eBPF 加速

NUMA 并行感知调度

【适用负载】多 NUMA 节点服务器

在多 Socket 服务器上部署内存数据库（如 Redis）、HPC 应用或大模型推理服务时，若未启用 NUMA 并行感知调度，进程可能因传统亲和性策略集中到少数几个节点，造成跨 NUMA 访问率上升、内存带宽受限、远端内存命中率下降，最终表现为吞吐量下降、长尾延迟上升。本模块的 opentunex-numa-sched-analysis 子技能会读取采集数据中的 NUMA 拓扑与跨节点访问率指标，判断当前负载是否落在 NUMA 不友好的访问模式上，若识别为瓶颈且系统支持 PARAL 内核特性，调优阶段会输出启用 PARAL 并设置 sched_util_low_pct 的具体建议，配合 numa_sched_tune.sh 调优脚本完成配置生效。

【预期收益】：跨 NUMA 访问延迟降低，内存带宽利用率提升，多 NUMA 负载吞吐提升 5%~20%。

窃取任务调度

【适用负载】高 CPU 利用率多核业务

在高 CPU 利用率的多核通用业务（如编译、并行计算、视频转码等）场景下，若 CPU 负载在物理核之间分布严重不均衡，会出现部分核满载而部分核空闲、%steal/sys 偏高、整体吞吐受限的现象。本模块通过 opentunex-stealtask-analysis 分析 CPU 负载均衡度与 SCHED_STEAL 内核配置，在判定为瓶颈且内核支持 STEAL 特性时，通过 opentunex-stealtask-tuning 输出启用 STEAL 窃取任务调度的具体建议。

【预期收益】：CPU 利用率均衡度提升，整体 CPU 利用率提升 10%~20%，任务调度延迟下降。

动态 SMT

【适用负载】低 CPU 利用率超线程业务

在启用 SMT（同时多线程/超线程）的服务器上，若业务整体 CPU 利用率较低，传统调度器仍会把同一工作单元的两个超线程分别调度到同一物理核的两个逻辑核上，造成超线程资源争抢、单线程延迟因邻居线程干扰而上升。本模块的 opentunex-dynamic-smt-analysis 子技能识别 CPU 利用率水平与 SMT 拓扑，结合内核 KEEP_ON_CORE 特性支持情况给出适用性结论；适用场景下由 opentunex-dynamic-smt-tuning 输出启用 KEEP_ON_CORE、让低负载工作单元独占物理核的具体配置。

【预期收益】：低负载场景下单线程延迟稳定性提升，超线程干扰消除。

Docker 算力统筹（burst）

【适用负载】容器化部署业务

在 Kubernetes/容器化部署环境中，业务通常以 cgroup cpu.cfs_quota_us 限制单容器 CPU 用量。当业务出现突发峰值时会触发容器 CPU 限流、任务被丢弃或排队等待，而宿主机上仍有空闲 CPU 资源未充分利用。本模块通过 opentunex-docker-coordination-burst-analysis 识别容器 CPU 限流与宿主机空闲情况，对支持 burst 内核特性的环境判定为适用；调优阶段由 opentunex-docker-coordination-burst-tuning 输出启用 sched_soft_runtime_ratio + cpu.soft_quota 的具体参数（如 ratio=20），让容器在突发期借用宿主机空闲算力。

【预期收益】：突发场景下容器 CPU 利用率提升，长尾延迟下降，资源利用率提升 10%~30%。

分域调度

【适用负载】多 NUMA 多实例容器/进程

在小配额多实例部署（如 Kubernetes 多副本）场景下，调度域粒度过粗容易导致进程在 NUMA 节点之间频繁漫游、跨 NUMA 调度抖动。本模块的 opentunex-soft-domain-analysis 结合 NUMA 拓扑与容器/进程部署情况，判定是否适合启用 SOFT_DOMAIN 分域调度特性；调优阶段输出启用分域调度的具体脚本。

【预期收益】：跨 NUMA 漫游抖动下降，调度延迟稳定性提升。

网卡多路径

【适用负载】多网卡多 NUMA 网络密集型业务

在多网卡多 NUMA 拓扑下，若中断亲和未做合理配置，会出现跨 NUMA 中断处理、网络抖动、单网卡打满而其他网卡空闲等问题。本模块通过 opentunex-multi-net-path-analysis 分析网卡、NUMA、IRQ 亲和配置，识别多路径调优适用性；适用场景下由 opentunex-multi-net-path-tuning 输出启用 oenetcls/venetcls 内核模块接管并配合 ntuple 配置的调优建议。

【预期收益】：跨 NUMA 网络中断亲和优化，网络吞吐与延迟稳定性提升。

BTB 分支预测

【适用负载】鲲鹏 920 关键线程业务

在鲲鹏 920 等新型 ARM 处理器上，CPU 的分支目标缓冲（BTB）资源按线程做了分区隔离，目的是减少跨线程分支预测相互污染。但在 redis、mysql 等线程数较多、关键线程切换频繁的业务上，线程分区反而限制了关键线程的 BTB 命中率，导致关键线程 IPC 下降。本模块的 opentunex-btb-analysis 在检测到 CPU 型号为鲲鹏 920 且关键进

程符合特征时，判定为适用；调优阶段由 opentunex-btb-tuning 给出禁用 TidCMP（BTB 线程分区）的 BIOS 调整建议。

【预期收益】：关键线程 BTB 命中率提升，单线程 IPC 改善，业务长尾延迟下降。

copy_from_user 拷贝优化

【适用负载】ARM64 大块读写热点业务

在 ARM64 Hisilicon CPU 上，若业务存在大块读写热点（如网络 IO、加密解密），传统 copy_from_user/copy_to_user 实现因未充分利用 ARM64 的双字加载指令（ldp/ldtp），会形成可观的性能瓶颈。本模块的 opentunex-copy-user-analysis 在 perf 热点分析中发现 copy_from_user/_to_user 占比较高且 CPU 型号符合时，判定为适用；调优阶段由 opentunex-copy-user-tuning 给出应用 ldp/ldtp 双字加载优化内核补丁并重新编译内核的具体方案。

【预期收益】：用户态/内核态拷贝吞吐提升，热点函数 CPU 开销下降。

hisock 网络加速

【适用负载】短连接高 PPS 网关/代理业务

在短连接、高 PPS 的网关、代理、NFV 业务中，nf_hook 是常见的热点路径，L2/L3 netfilter 框架的开销会显著影响 PPS 性能。本模块的 opentunex-hisock-analysis 在 perf 热点中识别到 nf_hook 占比较高时，判定为适用；调优阶段由 opentunex-hisock-tuning 给出基于 eBPF 的 hisock 加速方案，包含编译与加载 eBPF 程序的具体步骤，让数据包绕过部分 netfilter 路径。

【预期收益】：短连接 PPS 提升，L2/L3 netfilter 开销下降，网络吞吐与延迟改善。

6. 开发者工具链

DevStation 智能开发者桌面

DevStation 是基于openEuler的智能开发者工作站，专为极客与创新者而生。旨在提供开箱即用、高效安全的开发环境，打通从部署、编码、编译、构建到发布的全流程。它融合了一键式运行环境与全栈开发工具链，支持从系统启动到代码落地的无缝衔接。无需复杂安装，即可体验开箱即用的开发环境，通过新增MCP AI智能引擎，快速完成社区工具链调用，实现从基础设施搭建到应用开发的效率飞跃。

功能描述

开发者友好的集成环境：发行版预装了广泛的开发工具和 IDE，如 VS Codium 系列等。支持多种编程语言，满足从前端、后端到全栈开发的需求。

社区原生工具生态：新增 oeDeploy（一键式部署工具）、epkg（扩展软件包管理器）、devkit 和 openEuler Intelligence，实现从环境配置到代码落地的全链路支持。oeDevPluginss 插件+oeGitExt 命令行工具支持：专为 openEuler 社区开发者设计的 VSCodium 插件，提供 Issue/PR 可视化管理面板，支持快速拉取社区代码仓、提交 PR，并实时同步社区任务状态。 openEuler Intelligence 智能助手：支持自然语言生成代码片段、一键生成 API 文档及 Linux 命令解释。集成 epkg 包管理工具，能快速实现包括 Agent 等应用在 openEuler 环境上的安装部署体验。



图形化编程环境：集成了图形化编程工具，降低了新手的编程门槛，同时也为高级开发者提供了可视化编程的强大功能，预装 Thunderbird 等办公效率工具。

MCP智能应用生态构建：DevStation 深度集成 Model Context Protocol (MCP) 框架，构建完整的智能工具链生态，预装MCP智能工具链，支持oeGitExt、rpm-builder 等核心MCP Server，提供社区事务管理、RPM打包等能力，将传统开发工具（如Git、RPM构建器）通过MCP协议进行智能化封装，提供自然语言交互接口。

系统部署与兼容性增强：广泛的硬件支持，特别优化对主流笔记本/PC 硬件的兼容性（触摸板、Wi-Fi 、蓝牙），重构内核构建脚本（kernel-extra-modules），确保裸机部署体验。灵活部署形态，支持 LiveCD（一键运行无需安装）、裸机安装、虚拟机部署。

全新安装工具heolleo：heolleo 是一款专为 DevStation 设计的现代化客户端工具。其核心使命是简化DevStation的安装流程。采用模块化设计使其可以轻松扩展以支持不同的硬件架构（如 x86/ARM）、文件系统或引导加载器（GRUB等）。工具创新性地采用 SquashFS 安装方案，显著压缩标准镜像体积，有效降低开发者镜像下载成本。采用本地化安装：面向对稳定性与速度有极致要求，或需在无网络及受限环境中完成系统部署的用户，DevStation 提供本地 ISO 安装模式。该模式可充分复用现有系统镜像文件，带来高速、可靠且完全离线的安装体验。

应用场景

多语言开发环境：适用于需要同时开发多语言（如 Python、JavaScript、Java、C++）项目的开发者，无需手动配置环境，系统预装各种编译器、解释器和构建工具。

快速安装部署平台：DevStation 集成了 oeDeploy，可以对 kubeflow、k8s 等分布式软件实现分钟级部署，大幅减少开发者部署时间。oeDeploy 同时提供了统一的插件框架与原子化的部署能力，开发者只需遵循简单的开发规范，就可以快速发布自定义的安装部署插件，帮助更多用户解决安装部署问题。

面向测试/南向兼容性开发人员，提供硬件兼容性保障（支持主流笔记本/服务器），支持裸机部署测试驱动兼容性。

提升开发者效率：MCP RPM-builder工具链落地，提升MCP易用性，支持mcp servers自动化打包构建rpm包并发布到社区，确保mcp一键安装功能100%可用，构建完整的MCP智能应用生态REPO，覆盖部署、测试、性能调优等应用场景，包括：查询分配的社区issue，创建PR提交代码变更，通过CI/CD自动构建验证。

面向 AI/Agent 开发者，提供符合 Agent 工具链要求的标准化安装与部署环境，涵盖 Node.js 运行时版本管理、可观测性能力等基础环境支撑。

多发行版软件使用：开发者通常会受到宿主操作系统软件生态的限制，尤其 openEuler 环境安装应用依赖不满足场景，通过使用 epkg，可以分别创建绑定 openEuler、Debian 或 Alpine channel 的 environment，在保持宿主系统不变的情况下获取不同发行版的软件，快速实现包括 Agent 等应用在 openEuler 环境上的安装部署体验。

EPKG 软件包

epkg 是一个绿色轻量的跨平台包管理器，支持 Linux、macOS 和 Windows。 创建隔离的 enviroments，安装使用常见发行版（RPM、DEB、Alpine、Arch、Conda、Homebrew、MSYS2）的软件包，无需 root 权限。 每个环境绑定到一个 channel（如 Debian、Fedora、Alpine 等指定版本）。注册多个环境，将其二进制文件组合到 PATH 中，在一个 shell 中混合使用不同发行版的软件。

功能描述

epkg 采用用户态安装模式，软件包、环境及缓存可以存放在用户目录中，普通用户无需获得系统管理员权限即可完成软件安装、升级、删除和环境管理，降低对主机系统的修改范围。epkg 通过统一的 channel 机制对接不同软件生态，可在同一宿主系统中使用 openEuler、Fedora、CentOS、Debian、Ubuntu、Alpine、Arch Linux 等发行版的软件源，同时兼容 Conda、Homebrew 和 MSYS2 等语言及跨平台软件生态，这种机制能够将宿主操作系统和项目所需软件环境解耦，使软件依赖不再完全受限于宿主系统原生包管理器。



环境隔离是 epkg 的核心能力之一。每个 environment 可以绑定特定的 channel 和软件版本，不同环境之间的软件包相互独立，并可以通过 activate、register、unregister 等机

制进行切换和组合。用户可以同时维护多个相互独立的软件环境，例如针对不同项目分别创建 Python 3.10、Python 3.11 或 Python 3.12 相关环境，而无需频繁修改宿主系统中的 Python 版本；不同版本的编译器、运行时和系统工具也可以采用类似方式共存。通过环境导出与导入能力，还可以进一步实现开发环境的复制与分发。

epkg 进一步将操作系统级依赖与语言级依赖纳入统一环境管理。传统开发环境往往需要分别使用 apt、dnf、apk、pacman、Conda、pip 等工具安装不同层次的软件依赖，依赖关系分散在多个管理体系中。epkg 支持在同一 environment 中组合操作系统软件包与语言运行环境，使一个项目可以将编译器、系统库、Python 运行时以及其他开发工具统一组织。例如，一个 AI Agent 项目可能同时需要特定版本的 Python、Git、编译工具、系统库以及其他运行时组件，开发者可以将这些依赖放入同一环境进行管理，从而减少环境配置过程中的工具切换和依赖冲突。

在依赖解析与软件包管理方面，epkg 采用基于 SAT 的依赖求解机制处理复杂的软件依赖关系，并提供 install、update、upgrade、remove、list、search、info 等标准软件包操作。同时，epkg 提供事务历史和 restore 能力，在软件升级或环境发生异常后，可以依据历史状态进行恢复。软件存储采用文件级去重机制，并结合并行、分块下载和镜像机制降低重复下载带来的存储与网络开销。项目还提供约 14MB 级别的静态 musl 可执行文件，并支持 busybox-style applets，使其能够应用于资源受限的运行环境。

除传统包管理功能外，epkg 还提供从软件安装到运行环境隔离的扩展能力。epkg run 支持不同程度的隔离模式，包括基于用户命名空间和 bind mount 的环境隔离、基于 chroot 的文件系统隔离，以及基于 VM 的隔离方式。因此，epkg 的定位不仅是安装软件的工具，也可以作为构建轻量开发环境和隔离运行环境的基础组件。对于容器、嵌入式系统和本地 AI 开发等场景，epkg 能够在较小的系统开销下提供软件获取、依赖管理、环境组织和隔离运行能力。

应用场景

多发行版软件使用：开发者通常会受到宿主操作系统软件生态的限制。例如，服务器宿主系统为 openEuler，但某个项目依赖 Debian 生态中的软件包，另一个工具又主要通过 Alpine 或 Arch 生态提供。传统方式通常需要更换宿主环境、额外部署容器，或者手动处理软件包格式与依赖。使用 epkg 后，可以分别创建绑定 openEuler、Debian 或 Alpine

channel 的 environment，在保持宿主系统不变的情况下获取不同发行版的软件，并通过环境注册将所需工具加入当前 shell 的 PATH，实现多发行版软件的组合使用。

多版本共存与环境隔离：AI 开发尤其容易出现依赖冲突。例如，不同 Agent 项目可能分别依赖 Python 3.10、3.11 和 3.12，同时对 Node.js、编译器或系统库存在不同版本要求。如果直接修改系统 Python 或通过全局包管理器反复切换版本，容易影响已有项目。epkg 可以为不同 Agent 项目分别创建独立 environment，将对应版本的 Python 及其他依赖安装在各自环境中；需要运行某个项目时激活相应环境即可。不同环境的软件包相互隔离，项目之间不会因为版本升级产生直接影响，从而为多 Agent、多项目并行开发提供稳定的依赖基础。

操作系统依赖与多语言依赖统一管理：一个 AI 应用可能同时依赖 Python 运行时、Python 第三方库、Git、curl、编译工具、系统动态库以及其他命令行工具。传统方案通常需要组合使用系统包管理器、Conda 和 pip，并分别维护多份依赖配置。epkg 可以将 OS 级软件包和语言运行环境纳入同一个 environment，通过统一环境描述和导出机制组织项目依赖，使开发环境更容易复制到其他机器。对于需要频繁创建和销毁开发环境的 AI Agent 开发场景，这种“一套环境管理多层依赖”的方式可以降低环境初始化和维护成本。

无 root 权限的开发与计算环境：用户往往无法获得服务器或共享工作站的系统管理员权限，但仍然需要安装特定版本的软件。例如，在企业服务器、科研集群、共享 GPU 工作站中，普通用户可能需要独立安装 Python、Git、编译器或 AI 开发工具。epkg 能够将软件安装限制在用户自己的目录和环境中，不修改系统级软件配置即可完成依赖部署，因此适合权限受限的开发环境。openEuler 官方文档也明确提供了以非 root 用户为基础的 epkg 使用方式。

AI Agent 开发与本地运行环境：Agent 通常需要调用大量外部工具，例如 Python、Shell、Git、编译器以及各种命令行程序，同时不同 Agent 可能存在不同的软件版本和依赖要求。epkg 可以为每个 Agent 或 Agent 项目建立独立 environment，并结合`run`提供隔离执行能力，使 Agent 所需的软件依赖与宿主系统保持相对独立。例如，一个 Agent 需要 Python 3.11 和特定系统工具，另一个 Agent 需要 Python 3.12 及不同版本的运行时，可以分别构建对应环境，并根据任务选择环境执行。进一步结合文件系统或 VM 隔离能力，epkg 可以作为轻量级 Agent Sandbox 的底层环境管理组件，为 Agent 工具调用提供可控的软件供应与运行边界。

容器与嵌入式系统：传统发行版包管理器通常伴随较多系统组件和运行时依赖。当目

标是构建精简的容器或资源受限的系统镜像时，包管理器本身及其依赖可能占据额外空间。epkg 提供静态可执行文件以及面向 RPM、DEB、APK 等软件包生态的统一管理能力，可以作为容器中的轻量包管理工具，并通过软件存储去重和精简运行方式降低环境开销。项目仓库给出的设计目标包括在 RPM 和 DEB 类容器中减少包管理工具自身带来的额外空间占用。

总体而言，epkg 并非单纯替代 apt、dnf 或其他传统包管理器，而是围绕“软件来源统一、环境隔离、依赖统一和轻量运行”构建跨平台的软件环境管理能力。它将多发行版软件源、多版本软件、OS 级依赖与语言级依赖纳入统一的 environment 模型，并进一步向容器、AI 开发和 Agent 隔离等场景延伸，为开发者提供一种与宿主操作系统解耦的软件环境组织方式。

7. 著作权说明

openEuler 白皮书所载的所有材料或内容受版权法的保护，所有版权由 openEuler 社区拥有，但注明引用其他方的内容除外。未经 openEuler 社区或其他方事先书面许可，任何人不得将 openEuler 白皮书上的任何内容以任何方式进行复制、经销、翻印、传播、以超级链路连接或传送、以镜像法载入其他服务器上、存储于信息检索系统或者其他任何商业目的的使用，但对于非商业目的的、用户使用的下载或打印（条件是不得修改，且须保留该材料中的版权说明或其他所有权的说明）除外。

8. 商标

openEuler 白皮书上使用和显示的所有商标、标志皆属 openEuler 社区所有，但注明属于其他方拥有的商标、标志、商号除外。未经 openEuler 社区或其他方书面许可，openEuler 白皮书所载的任何内容不应被视作以暗示、不反对或其他形式授予使用前述任何商标、标志的许可或权利。未经事先书面许可，任何人不得以任何方式使用 openEuler 社区的名称及 openEuler 社区的商标、标记。

9. 附录

附录 1：搭建开发环境

环境准备	地址
下载安装 openEuler	https://openeuler.org/zh/download/
开发环境准备	https://atomgit.com/openeuler/community/blob/master/zh/contributors/prepare-environment.md
构建软件包	https://atomgit.com/openeuler/community/blob/master/zh/contributors/package-install.md

附录 2：安全处理流程和安全批露信息

社区安全问题披露	地址
安全处理流程	https://atomgit.com/openeuler/security-committee/blob/master/docs/zh/vulnerability-management-process/security-process.md
安全披露信息	https://atomgit.com/openeuler/security-committee/blob/master/docs/zh/vulnerability-management-process/security-disclosure.md
安全保障策略总纲	https://atomgit.com/openeuler/security-committee/blob/master/security-strategy-overview.md