

大模型公共服务平台建设

使用开源软件的实践和踩坑记录

王永福

中国科学技术大学

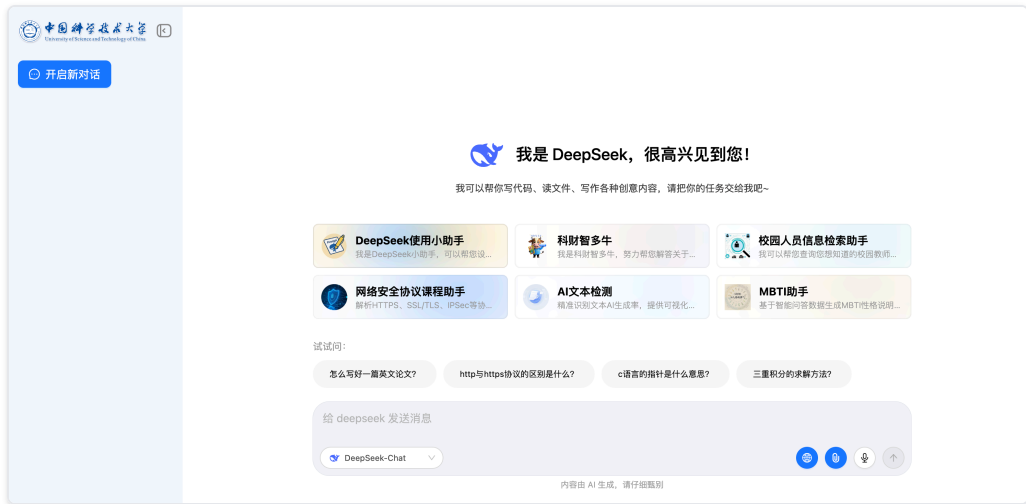
目录

- 01 背景：USTC Chat
- 02 架构演进：LiteLLM → 管理平台 → Dynamo
- 03 踩过的坑：网络、编译、开源软件 Bug、协议细节
- 04 现状：算力与服务规模
- 05 体会与未来展望

背景

从大模型公共服务平台的前身说起

前身：USTC Chat



chat.ustc.edu.cn

2025 年初上线

DeepSeek R1

形态：仅提供 Chat Web UI，用户对话上下文大多很短

问题：纯粹的 Chat 形态，无法支撑正在兴起的 **Agentic Workload**（长上下文、工具调用、多轮交互）

需要一个面向全校、可编程接入的大模型公共服务平台 →

架构演进

从统一网关到推理调度

第一版：统一服务体系

2025 年 10 月

需求：支持多种大模型、统一接入、分发 Virtual Key、用量统计

架构：业务网关 → 推理引擎

- 业务网关负责鉴权和计费
- 选型 **LiteLLM**:
 - 满足功能要求，自带 UI
 - Web API 完备，支撑后期自动化

用户少 → 直接在 LiteLLM 上配置 Virtual Key，没有面向用户的页面



模型：

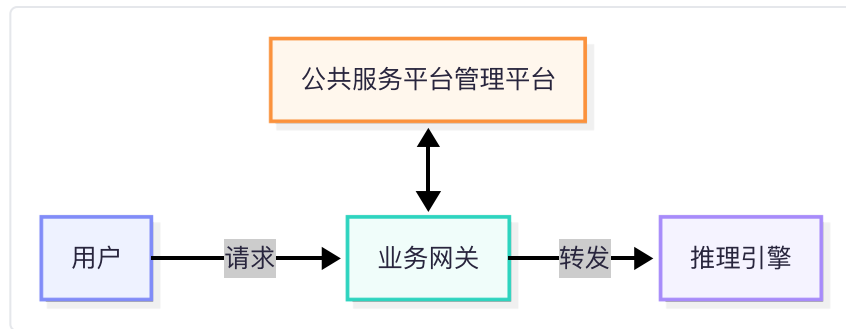
DeepSeek V3 / R1

公共平台开放

2026 年 3 月

背景：OpenClaw 爆火，需要把 LLM 服务开放给全校师生

- 新增**管理平台**：面向用户和管理员，提供申请 / 审批等功能
- 管理平台调用业务网关 API 完成配置（Virtual Key 签发、模型授权等）
- 业务网关（LiteLLM）专注**数据平面**：鉴权、计费、请求转发
- 推理侧仍是 LiteLLM 直连多个 vLLM / SGLang 实例做负载均衡



引入 Dynamo 2026 年 4 月

背景：东风云二期引入大量华为设备，新增多个 Qwen / DeepSeek 推理引擎实例

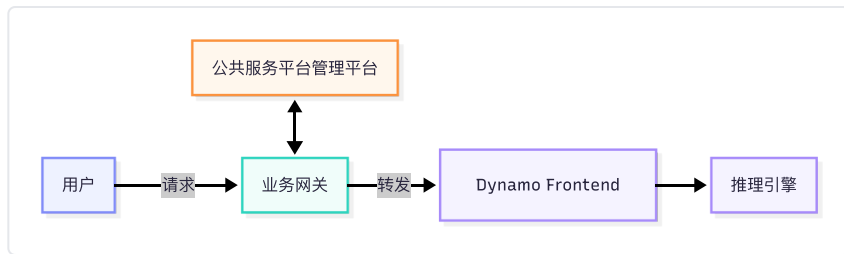
痛点

- 每个引擎都要在 LiteLLM 上配置为上游，非常麻烦；引擎变化需手动改配置
- LiteLLM 随机选择上游，无法感知 **Prefix Cache** 进行调度

方案：引入 Dynamo 作为推理引擎前端

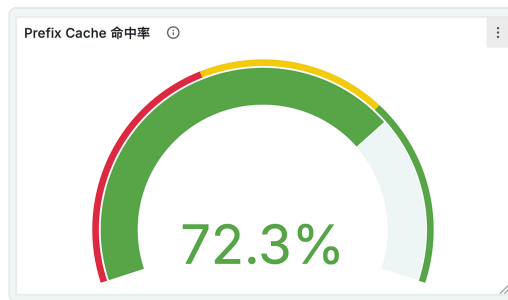
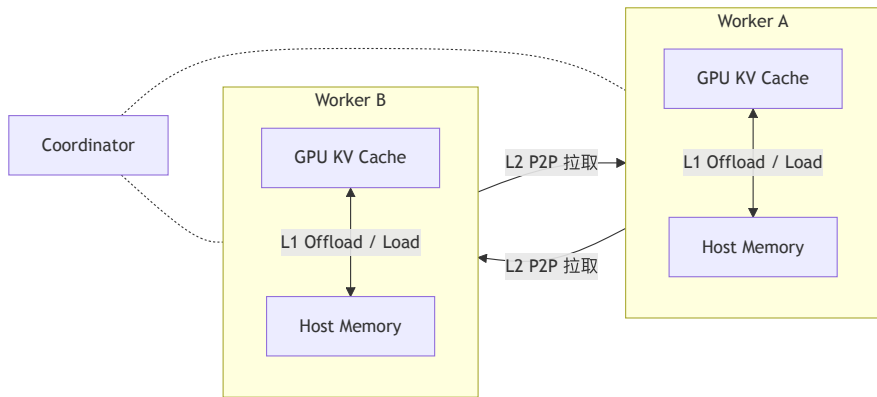
- 基于 etcd 自动发现 worker：每个模型在 LiteLLM 中只需配置一个上游
- 支持 **Prefix Cache** 感知调度
- Frontend 负责 Tokenization / Chat Template 渲染 / 解析，引擎只收发 Token ID
- 渲染解析与引擎**解耦**：修相关 bug 不用动推理引擎
- 魔改了一套 Dockerfile，在 vllm-ascend 中加入 Dynamo
-> 可用，但 vllm-ascend 的 KV 事件上报仍有问题

当前架构

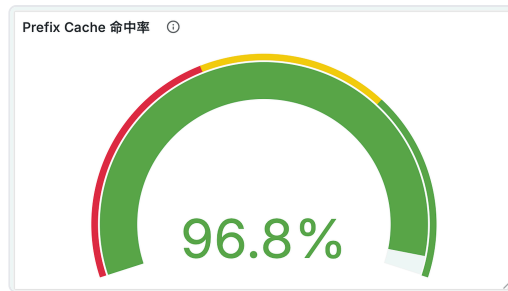


性能优化：LMCache

Dynamo 只解决了 **Prefix Cache** 感知的路由，还需要 KV Cache 池化和多级存储 → 引入 **LMCache**: L1 Host Memory Offload + L2 KV P2P



引入前: Prefix Cache 命中率 72.3%



引入后: Prefix Cache 命中率 96.8%

性能优化：GLM-5.2 DCP

问题

- GLM-5.2: 744B 参数, 支持 1M 上下文
- TP=16 下单卡权重占用 54GB, KV Cache 仅剩 10GB+
- 仅能支持 250K (TP=16) / 500K (TP=8, PP=2) 上下文

机会

- GLM-5.2 采用 MLA, 仅一个等效 KV 头, 但 TP 下各 Rank 存储**重复内容**, 浪费空间
- vLLM 的 **DCP** (Decode Context Parallel) 将 KV Cache 按 Token 切分存储

但 DCP 不支持 DeepSeek Sparse Attention

- 分析发现 FlashMLASparse 算子输出含 **log-sum-exp** 项
→ Attention 可合并, 只需引出该项输出
- DSA 需要全局 Top-2048, DCP 各 Rank 缺少全局视野
→ 每 Rank 先选 Local Top-2048, AllGather 后再选全局 Top-2048
- **FP8 支持**: DSA FlashMLA 实现中 Q 和 Attention 输出均为 BF16, 没有 FP8 moving part
→ 修改判断即可让 DCP 直接支持 FP8 KV

回馈上游

向 vLLM 提交 **PR #46178**，让 DSA（稀疏注意力）跑在 DCP 上：

- 约 10 个 commit，覆盖 DCP 支持、heads count、AllGather 融合、MTP、persistent top-K、FP8 KV 等

虽然并没有人理我

Commits on Jun 20, 2026	
DSA: add DCP support	RTXUX committed on Jun 20
DSA: fix DCP	RTXUX committed on Jun 20
FlashMLA: fix DCP heads count	RTXUX committed on Jun 20
DSA: fix hang in DCP	RTXUX committed on Jun 20
DSA: use single AllGather and fuse some operations	RTXUX committed on Jun 20
DSA: re-enable persistent top-K	RTXUX committed on Jun 20
DSA: fix MTP in DCP	RTXUX committed on Jun 20
DSA: optimize MTP path	RTXUX committed on Jun 20
DSA: support FP8 KV	RTXUX committed on Jun 20 · 6 / 8

性能优化：DSpark 自适应投机长度 问题

vLLM 0.27.2rc0 引入 **DSpark 动态投机长度** (PR #47808)

验证阶段的瓶颈

- DSA indexer 在投机解码验证阶段调用 DeepGEMM 的 **paged MQA logits kernel**
- 自适应验证在设备端按置信度**裁剪每个请求的 draft token 数**
→ 要求 kernel 支持 **varlen** (per-request 长度从 device tensor 读取)

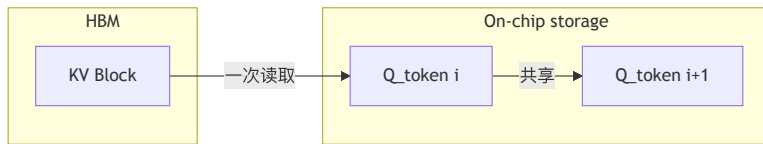
但 DeepGEMM 只在 SM100 上实现了 varlen 版本

Hopper 已经是昨日黄花子

性能优化：DSpark 自适应投机长度 实现

DeepGEMM 的 `nv_dev` 分支已有 SM90 varlen 所需的 scheduler 基础设施

- 移除相关 Guard
- 补齐缺失的部分
- 按 **2-token atom** 组织流水线：同一序列的相邻 token 配对，共享一次 KV 读取



✓ 在没有官方支持的情况下，为 SM90 补出了 varlen 版本 kernel

性能优化：DSpark 自适应投机长度 收益

Kernel 层面 (SM90 varlen vs flattened)

```
SM90 varlen vs flattened (non-varlen next_n=1) paged MQA logits:
seqs= 256 tok/seq= 1 avg_kv= 8192 tokens= 256 | varlen: 111.1 us | flat: 108.1 us | speedup: 0.97x | KV traffic ratio (flat/varlen ideal): 1.00x
seqs= 256 tok/seq= 2 avg_kv= 8192 tokens= 512 | varlen: 113.6 us | flat: 203.7 us | speedup: 1.79x | KV traffic ratio (flat/varlen ideal): 2.00x
seqs= 256 tok/seq= 4 avg_kv= 8192 tokens= 1024 | varlen: 212.9 us | flat: 396.9 us | speedup: 1.86x | KV traffic ratio (flat/varlen ideal): 4.00x
seqs= 256 tok/seq= 8 avg_kv= 8192 tokens= 2048 | varlen: 405.1 us | flat: 767.0 us | speedup: 1.89x | KV traffic ratio (flat/varlen ideal): 8.00x
seqs= 256 tok/seq= 1 avg_kv= 32768 tokens= 256 | varlen: 395.9 us | flat: 391.1 us | speedup: 0.99x | KV traffic ratio (flat/varlen ideal): 1.00x
seqs= 256 tok/seq= 2 avg_kv= 32768 tokens= 512 | varlen: 402.8 us | flat: 763.0 us | speedup: 1.89x | KV traffic ratio (flat/varlen ideal): 2.00x
seqs= 256 tok/seq= 4 avg_kv= 32768 tokens= 1024 | varlen: 796.5 us | flat: 1528.0 us | speedup: 1.92x | KV traffic ratio (flat/varlen ideal): 4.00x
seqs= 256 tok/seq= 8 avg_kv= 32768 tokens= 2048 | varlen: 1559.0 us | flat: 3033.0 us | speedup: 1.95x | KV traffic ratio (flat/varlen ideal): 8.00x
```

- 验证 batch 在 **tok/seq ≥ 2** 时提速 **1.8 ~ 1.9x** (KV 读取量近似减半)
- tok/seq = 1 时与原版持平

端到端

DeepSeek-V4-Flash + DSpark + 自适应验证长度，在 **SM90** 上可用

DP8 + EP 实测吞吐

+10 ~ 13%

2303 vs 2087 tok/s

架构演进历程



踩过的坑

给 self-host LLM 的同学们一些参考

坑 1: RoCE ARP (上)

ARP 应答错乱

- Linux 默认响应目标为本机**任意 IP** 的 ARP 请求
- 集群每台机器 8 块 HCA 在同一个广播域 → ARP 响应可能从**错误的网卡**发出
-  方案: `arp_ignore=1`, `arp_announce=1`

ARP 请求风暴

- 每台机器 8 卡需与本机及其他机器的 8 卡建连: $N(8 \times 7 + (N - 1) \times 8) = 8N^2 + 48N$ 个 ARP 请求
- 可能导致莫名其妙的 throttle, 部分 rank 地址解析超时
-  Workaround: Linux 启用 GARP; 交换机启用 ARP 学习和代答, HCA 的 ARP 请求在交换机直接处理
-  正经方案: 按 rail 划分广播域 (太麻烦没做)

坑 1: RoCE ARP (下)

内核 6.18 的新行为

Linux 6.18 (commit `c31e403`) 之后, 对于目的地址路由类型为 `local` 的 RDMA 地址解析请求, **直接返回 destination MAC = source MAC**, 而不实际执行地址解析。

后果

- 解析本机其他 rank 时命中 local 路由
- 拿到的是自己的 MAC, 根本未真正解析
- 建连失败

✅ 方案

- 把每个 HCA 划进一个单独的 **VRF**, 各自独立路由表
- 本机其他 rank 不再匹配 local 路由
- 走正常 ARP 解析

坑 2：鲲鹏 920 没有 RCpc

华为平台使用鲲鹏 920 CPU (TaiShan v110 核心)

- qemu-user arm64 编译出来的二进制 → **SIGILL**
- 改在本机编译 → build script 自己 **SIGILL**
- 原因：Dynamo 默认启用 `ldapr` (ARMv8.3-A **RCpc** 扩展)，鲲鹏 920 未实现
-  解决方案：编译目标改为 **ARMv8.2-a**

```
- # Target Graviton 2 / Ampere Altra + machines. NEON vector, and LSE a
- # Target Graviton 2 / Ampere Altra / Kunpeng 920 + machines. NEON vec
- # -rcpc is required because LLVM enables ldapr (ARMv8.3-A RCpc) by de
- # neoverse-n1, but Kunpeng 920's TaiShan v110 core does not implement
- [target.aarch64-unknown-linux-gnu]
- rustflags = ["-C", "target-cpu=neoverse-n1", "-C", "force-frame-point
- rustflags = ["-C", "llvm-args=-march=armv8.2-a", "-C", "force-frame-p
```

坑 3: LiteLLM Bugs

① 下游断开连接时，LiteLLM 不会取消上游请求！

[Bug]: LiteLLM Proxy doesn't cancel ongoing request when client connection closes #22805

✓ Closed

#30223

官方有个 Completed 的 issue 和 Merged 的 PR —— 但实际上根本没修好！

② `max_parallel_requests` 的 Redis Key TTL 只有 60s

如果请求处理超过 60s，用户可以在 60s 后继续发起新请求，突破并发请求数限制

坑 4: DeepSeek 的 Chat Template

模型服务时，客户端发来的消息列表（system / user / assistant / tool）要先由 **Chat 模板** 拼成一段 prompt 再送进模型。

- DeepSeek 的模板对 system message 的处理是**直接插入**，后面不接任何东西
- Claude Code 某次更新后，会在对话**中间甚至末尾**插入 system message → 渲染后末尾缺少 Generation Seed → 生成异常

```
3 ---
4
5 ### 10. Tool Message Merging
6
7 DeepSeek V4 does not have a standalone "tool" role. The `n
  OpenAI-format conversations (with separate "tool" role m
  as `content_blocks` with `type: "tool_result"`, encoded as
  tool_result><| Assistant |><think>
8
9
```

正常渲染（末尾有 Generation Seed）

```
## content
`lib/llm` wires this crate in via `crate::prep
lib/llm-local glue that can't live here (the `
`MediaRequestExt` extension trait, and `prompt
else imports `dynamo_renderer` directly.
...
```

system 插入末尾，渲染异常

```
## Chunk Timeline
```text
00000 choice=0 content: </
00001 choice=0 content: | DSML |
00002 choice=0 content: tool
00003 choice=0 content: _c
00004 choice=0 content: alls
00005 choice=0 content: >
```
```

异常输入 → 异常输出

✅ 方案：将非开头的 system message 转为 user message

坑 5: Qwen 工具调用参数的两派写法

按 OpenAI 协议, assistant 消息里工具调用的 `arguments` 是一个 **JSON 字符串**。多轮对话中客户端会把上一轮 assistant 消息原样带回, 每次渲染历史都会遇到这个字符串。

不同模型训练数据里工具调用的历史写法分两派:

XML 派 (Qwen3-Coder、Qwen3.6/3.8)

```
<parameter=command>
echo hello
</parameter>
```

JSON 派 (老 Qwen3、Hermes)

```
"arguments": {"command": "echo hello"}
```

- **XML 派的问题:** `<parameter>` 标签只能对「已解析的对象」逐项生成, 而模板引擎 (Jinja) 无法解析 JSON 字符串
→ 服务端必须在渲染前把字符串解析成对象, 否则整个字符串原样塞进历史
- Dynamo 的启发式检测只看模板里有没有 `arguments is string` 分支
新的 XML 派模板带有这样的兜底分支 → **误判为 JSON 派** → JSON 解析被关闭
-  **Workaround:** 增加环境变量 `override`

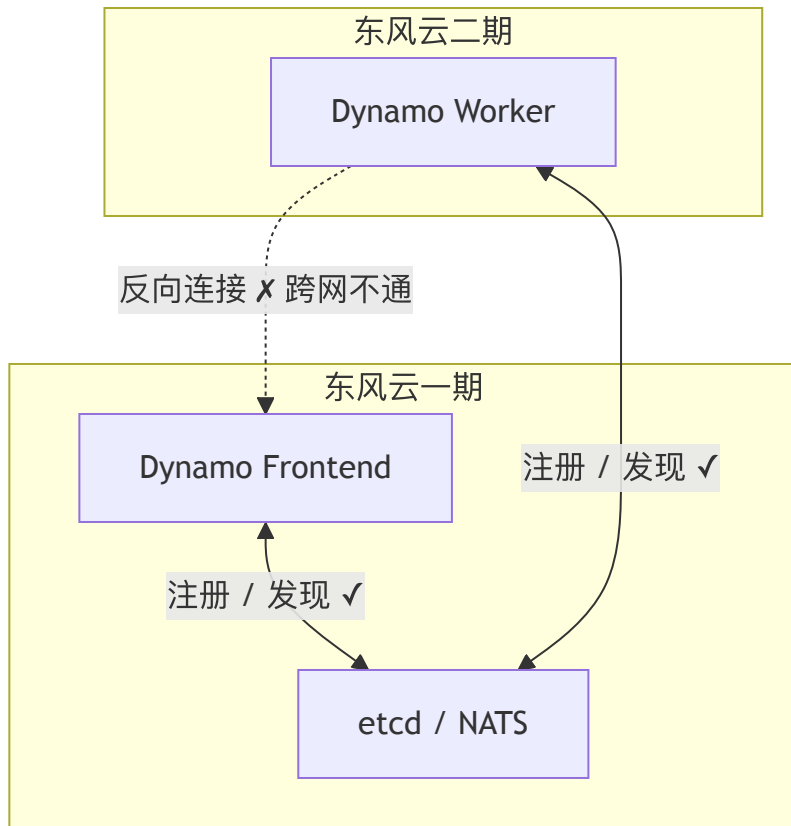
 **错误渲染:** JSON 字符串被原样拼进历史

→ 多轮对话后工具调用生成异常

```
<function=run>
{"command": "echo hello"}
</function>
```

坑 6：跨集群与反向连接

- 东风云一期和二期属于**不同内网**；LiteLLM 在一期，希望把 Dynamo Frontend 全部部署在一期
- Dynamo Request Plane 有 NATS 选项（虽已Deprecated）
我们以为 frontend 和 worker 只要能连到 etcd 和 NATS 就行
- 实际处理请求时，**worker 会向 frontend 发起反向连接** → 跨集群部署失败
- 结论：frontend 和 worker 需要能用对方 IP **直接互连**
- ☒ Workaround：把 frontend 移到二期内网



坑 7: FlashInfer Sampling Kernel 卡死

- **现象:** vLLM + LMCache (GLM-5.2 / DSv4-Flash) 高负载下引擎崩溃
 - `TimeoutError: RPC call to sample_tokens timed out` (300 s 执行超时)
 - LMCache 进程不崩溃但随之僵死, 显存残留导致 vLLM 无法重启
- **根因:** 跨 CTA 软件屏障 + 非协作式启动
 - top-k 采样使用多 CTA radix kernel, 组内 CTA 通过全局内存中的 `arrival_counter` 自旋屏障同步
 - Barrier 工作的前提: **同组所有 CTA 必须同时驻留 SM**
 - 但 Kernel 用普通 `cudaLaunchKernel` 启动 —— CUDA 从不保证共同驻留 (网格大小按"每 SM 一个 CTA"估算, 只是启发式假设)
- **触发条件:** LMCache MP server、NCCL/AgRs 集合通信、MTP draft kernel 争抢 SM (含 MPS 跨进程争抢)
 - 负载越高越易触发; 组内 CTA 分裂到两波调度后, 驻留者自旋不释放 SM, 未调度者无法被调度 → **流永久卡死, 无超时、无自愈**
-  **方案:** 设置 `VLLM_USE_FLASHINFER_SAMPLER=0`, 禁用 FlashInfer 采样器

现状

现状：算力

DeepSeek V4 Flash

算力占比 37.5%

GLM 5.2

算力占比 25.0%

DeepSeek V4 Pro

算力占比 25.0%

Qwen3.8-27B

算力占比 12.5%

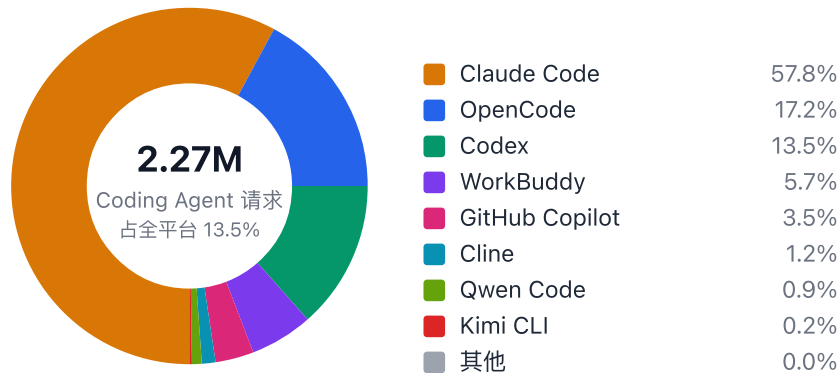
现状：服务

- 日活用户：500 ~ 1000
- 日处理 Token：10B+
- 日请求数：~500K

主要用途

- 个人自由探索
- 支撑校内应用

审计
财务问答
入校报备
.....



体会与展望

体会

LLM Serving Stack 博大精深

网关、调度、KV Cache、并行策略、协议处理、模板渲染.....
每一层都有学问

模型厂有很多未公开的内容

Serving 侧的细节（如 Protocol Handling/Rendering/Parsing）
只能靠实测摸索

开源软件好

遇到 bug 能自己修，需要的功能能自己加

LLM Serving 开源项目普遍缺乏测试

LLM Serving Stack 项目的测试覆盖薄弱，升级和改动都需要
自行验证

买最新最好的硬件

Hopper 已经是昨日黄花 新功能（如 DeepGEMM varlen）往往优先支持最新架构，老架构只能靠自己补

未来展望

完善 Observability 建设

Metrics / Tracing / 日志全链路覆盖

全平台 Kubernetes 管理

统一编排与调度，降低运维成本

持续优化性能

KV Cache 池化 / 持久存储
请求调度
算子优化

思考：模型分析自己的 Parser

现象：用 DeepSeek 分析 DeepSeek 的 DSML parser 时，模型输出与 tool call 会发生混乱

Use vs Mention（使用 vs 提及）

哲学上区分「使用一个词指称世界中的事物」与「提及这个词本身」：

- **使用：**Paris is the capital of France.（词指向现实中的城市）
- **提及：**"Paris" has five letters.（词指向它自身这串字符）

LLM 输出工具标签去执行动作，是**使用**符号作为操作指令；解释 parser 如何工作，是**提及**符号作为字面字符串。人类写作有引号、斜体消歧；LLM 推理中若没有带外信令或独立的 token ID，parser 无法区分一个标签是被使用还是被提及。

Technical Mitigation

1. **Tool Registry Validation：**只识别已注册的工具标签，未注册的按普通文本处理
2. **Code Block Fencing：**代码块 / 引用块内部不触发 tool call 解析
3. **专用 Token / 带外信号：**工具标签使用独立 token ID，而非可打印字符串

自指的边界条件

数理逻辑中，**哥德尔编码**把关于公式本身的元数学命题编码进公式内部，揭示了系统在自反时非常容易陷入自指悖论。

计算机与哲学中的 **Quine**：不读取输入就输出自身源码的程序。

当生成模型反思支配其运行时的解析规则，它是否正好撞上这个自指边界条件？

观察到的相似问题

和 agent 讨论上下文压缩的机制与实现，然后执行 /compact：它有概率输出一个**模拟的压缩结果**，而不是真正输出自己上下文的压缩版——同样是 use vs mention 的混淆。

谢谢！

Q & A