INTERVIEW PREP · SYSTEMS NOTES

Inference Serving 全景

从指标、瓶颈到框架与生态

LLM · MoE · 大规模集群 · 推理框架 · 后训练 · Omni · 扩散 · 机器人 · Coding Agent · AI for Science · 源码导读



作 者

kzhoulatte

2026 年 10 月

第 1 版

CONTENTS

目录

· 阅读地图与声明 3

01 推理服务本身：要求与指标 *指标港 · TTFT · goodput* 4

02 物理限制：为什么 LLM 推理难 *翻越 Roofline 山脉* 9

03 需要解决的问题 *瓶颈沼泽* 13

04 LLM serving 的核心技术 *王都* 15

4.1 批处理与调度：continuous batching 15

4.2 KV cache 管理：PagedAttention、前缀缓存、分层存储 16

4.3 Chunked prefill：把 prefill 切碎塞进 decode 批 18

4.4 Prefill–Decode 分离（PD disaggregation） 18

4.5 投机解码（Speculative decoding） 19

4.6 Kernel 与 runtime：把每一步做快 21

4.7 量化：少读字节、多放 KV 24

4.8 模型结构侧的优化：从源头减小 KV 与计算 24

4.9 并行策略：模型放不下、或一张卡太慢 25

4.10 结构化输出、多 LoRA、API 层功能 26

4.11 总表：技术 → 问题 → 代价 26

05 扩展到 MoE：显存按总参数、计算按激活参数 *专家群岛* 28

06 扩展到大规模集群 *集群高原* 32

07 LLM 推理框架 *框架城邦：vLLM · SGLang* 35

08 后训练里的推理：RL rollout 是一个推理问题 *RL 环礁* 38

09 Omni / 语音 / 多模态模型的推理服务 *多阶段湖区* 42

10 扩散模型推理 *去噪湾* 45

11 机器人推理：从“吞吐”回到“实时” *实时岬* 49

12 Coding Agent 的推理：长会话、高命中、慢工具 *Agent 港 · 长会话* 53

13 AI for Science 的推理：结构、几何与物理 *科学群岛* 56

14 和 inference serving 相关的技术 *Sandbox 要塞* 59

15 公司与开源项目地图 *生态港* 63

16 源码导读：带着问题读代码 *藏书阁 · Code Wiki* 66

17 面试速查 *灯塔* 69

18 参考文献与延伸阅读 *图书馆* 74

阅读地图

这是一张“推理服务大陆”的地图：西岸的港口是起点（指标），翻过 Roofline 山脉（物理限制）、穿过瓶颈沼泽（要解决的问题），到达王都（LLM 核心技术）；向东是 MoE 群岛和集群高原；海上的岛屿是换了负载之后的世界：后训练、Omni、扩散、机器人、Coding Agent、AI for Science；沿西海岸北上是 Sandbox 要塞、生态港、藏书阁（源码导读）和面试灯塔。红色虚线是推荐航线。点击任意地点，章节会在右侧打开。

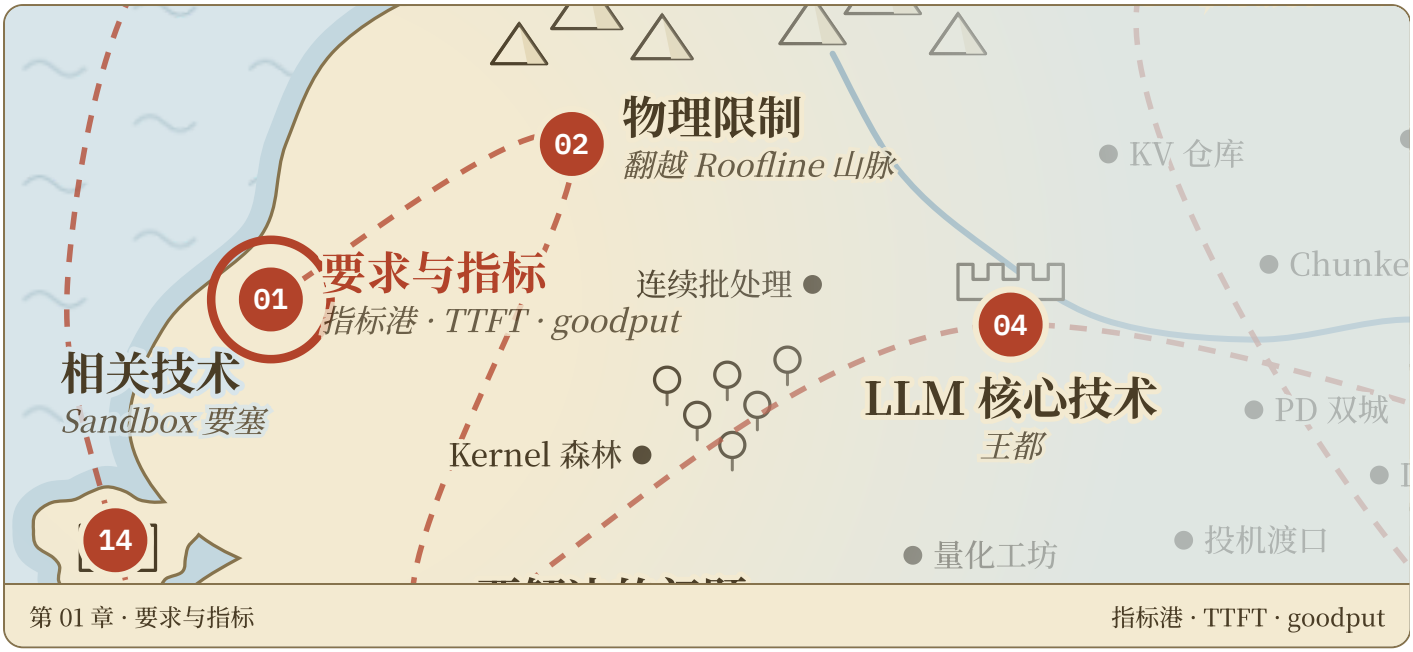


01 章节（点击打开） ● 小节 ---- 推荐阅读航线

作者：kzhoulatte · 2026 年 10 月 · 第 1 版

声明。本文由 kzhoulatte 编写整理，用于个人学习与面试准备。内容基于公开的论文、官方文档与开源代码，文中的归纳、取舍与观点仅代表作者本人，不代表任何雇主或开源项目。行业与生态信息截至约 2026 年中，可能已经变化；文中源码链接均固定在 2026-10-31 前后的 commit，示意图中的曲线与数字除注明出处者外均为示意。引用或转载请注明作者与出处。

阅读方式。每章开头的地图截面标出该章在全景图中的位置；正文中的灰色胶囊是可点击的源码链接，◈ 标记的是 Google Code Wiki 链接。



01 推理服务本身：要求与指标

Inference serving 的任务是把训练好的模型变成一个服务：接收请求、在加速器上跑前向、把结果（常常是流式地）返回。它要同时满足三类目标，而这三类目标互相拉扯：

延迟（用户体验）

第一个 token 多快出来，之后每个 token 隔多久，整条回答多久结束。看的是 p50 / p90 / p99，不是平均值。

吞吐与成本（经济性）

每块 GPU 每秒能产出多少 token，折算成每百万 token 的成本。批越大吞吐越高，但单请求越慢。

质量与可靠性

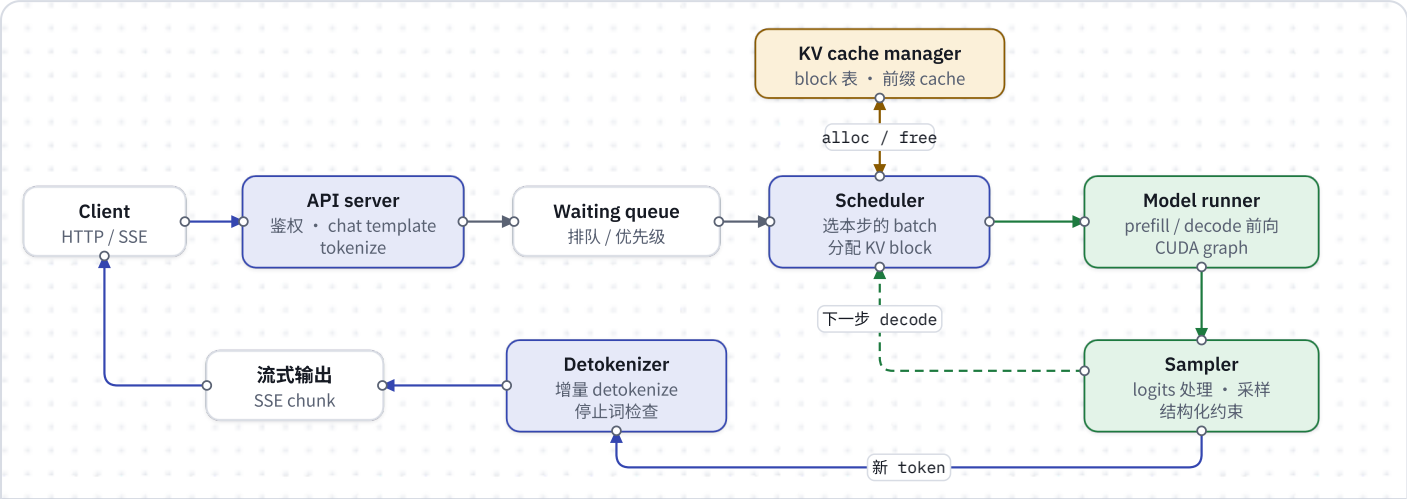
量化、投机解码、近似 cache 不能让输出变差；服务要可用、可扩缩、能抗突发流量、能隔离租户。

四种典型场景

场景	例子	最在意	可以牺牲
在线交互	Chat、Copilot 补全	TTFT、TPOT 的 p99，流式输出	一部分吞吐（批不能太大）
Agent / 长上下文	编码 agent、多轮工具调用、RAG	长 prompt 的 prefill 速度、前缀 cache 命中率、端到端完成时间	单 token 延迟可略放宽
离线批处理	数据合成、评测、Batch API、RL rollout	总吞吐、每 token 成本	延迟几乎无所谓（RL 例外：长尾会拖慢整个 step）
实时流式	语音对话、同传、机器人控制	首包延迟、稳定的输出节奏（不能卡顿）、抖动	峰值吞吐；常常 batch=1

一个请求的生命周期

以 OpenAI 兼容的 `/v1/chat/completions` 流式请求为例。前端进程做 HTTP、chat template、tokenize；调度器决定这一步让哪些请求上 GPU；model runner 跑前向；sampler 采样；detokenizer 把 token 变回文本，经 SSE 流回客户端。

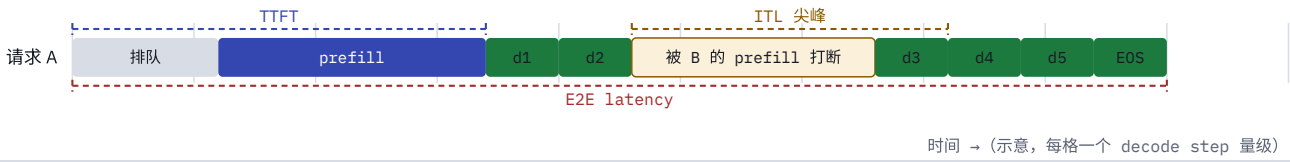


请求路径。绿色虚线是每个 decode step 的循环：调度 → 前向 → 采样 → 流式返回，直到 EOS 或 max_tokens。CPU 侧（API、tokenize、调度）若慢，会直接表现为 GPU 空转。

核心指标定义

指标	全称	定义	主要受什么影响
TTFT	Time To First Token	请求到达 → 第一个输出 token 返回。包括排队、prefill、第一次采样。	prompt 长度、排队、prefill 算力、前缀 cache 命中
TPOT / ITL	Time Per Output Token / Inter-Token Latency	TPOT = (E2E − TTFT)/(n_out − 1)，是单请求的平均值；ITL 是相邻两个 token 的间隔分布，能看出卡顿。	decode 批大小、显存带宽、被 prefill 打断
E2E latency	End-to-end latency	请求到达 → 最后一个 token。≈ TTFT + (n_out − 1) × TPOT。	输出长度（推理模型 CoT 很长）
Throughput	吞吐	输出 token/s、总 token/s（含输入）、请求/s。常按每 GPU 归一化。	批大小、kernel 效率、显存能容纳多少并发
Interactivity	tokens/s/user	单个用户看到的生成速度 = 1/TPOT。和每 GPU 吞吐构成 Pareto 曲线。	同上，方向相反
Goodput	满足 SLO 的吞吐	只统计 TTFT 和 TPOT 都满足 SLO 的请求的吞吐（DistServe 提出的口径）。比裸吞吐更接近“能卖的产能”。	尾延迟、P/D 干扰
SLO attainment	SLO 达成率	满足 SLO 的请求占比，例如 “99% 请求 TTFT < 2 s 且 TPOT < 50 ms”。	负载、调度策略

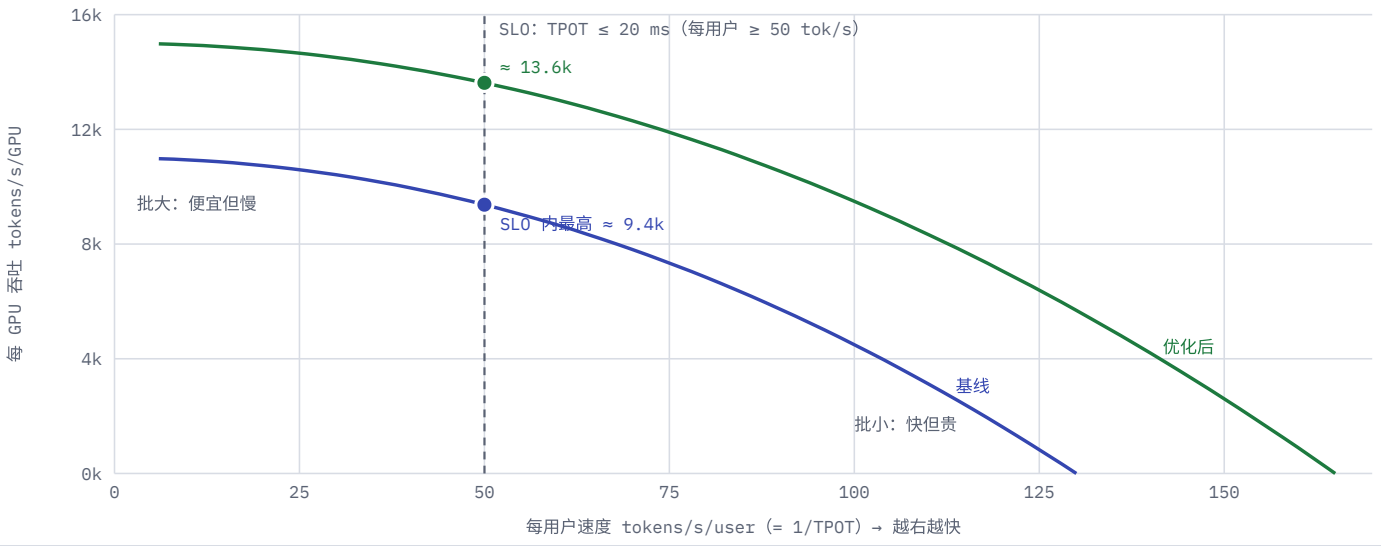
指标	全称	定义	主要受什么影响
MFU / MBU	Model FLOPs / Bandwidth Utilization	MFU = 实际 FLOP/s ÷ 峰值 FLOP/s（prefill 看它）；MBU = 实际字节/s ÷ 峰值带宽（decode 看它）。	kernel 质量、批大小、CPU 开销
$/1M tokens	单位成本	GPU 小时价 ÷ (每 GPU 每小时产出 token / 10^6)。输入、输出、缓存命中的输入通常分别计价。	吞吐、硬件价格、利用率（空闲时间也要付钱）
其他运营指标		排队时间、抢占 / 重算次数、前缀 cache 命中率、KV 使用率、错误率、冷启动时间、可用性。	



单个请求的时间线。TTFT 包含排队 + prefill；之后每个 decode step 产出一个 token。第 3 个 token 前那段长间隔是被别人的 prefill 打断（ITL 尖峰），这就是 chunked prefill 和 PD 分离要解决的问题。

吞吐和延迟的权衡：Pareto 曲线

decode 一步的耗时近似是 $t(B) \approx t_0 + c \cdot B$：$t_0$ 是把权重读一遍的固定成本，$c \cdot B$ 是随批大小增长的部分（读各自的 KV、计算）。于是批大小 B 越大，每 GPU 吞吐 $B/t(B)$ 越高，但每用户速度 $1/t(B)$ 越低。SemiAnalysis 的 InferenceMAX 等基准就是把这条曲线画出来比较。



示意曲线（非实测）。曲线上每个位置对应一个批大小：左上是大批（吞吐高、每用户慢），右下是小批。SLO 要求每用户 ⩾ 50 tok/s（TPOT ⩽ 20 ms）时，应选 SLO 线右侧吞吐最高的点；优化（更好的 kernel、FP8、投机解码）把整条曲线往右上推。

两条要背下来的关系式

```
E2E ≈ TTFT + (n_out − 1) × TPOT
并发数 L = 到达率 λ × 平均停留时间 W            # Little's law
例：每秒 20 个请求，每个平均 15 s 才结束 → 系统里稳态约 300 个并发请求，KV cache 要装得下这 300
```

SLO 的常见量级（经验值，因产品而异）

- 人的阅读速度约 4–8 token/s，所以聊天的 TPOT 在 30–100 ms 之间体验上差别不大；但 agent 和推理模型输出很长，TPOT 直接决定完成时间，所以“每用户速度”越来越受重视。
- 语音对话要求首包音频在几百毫秒内出来；代码补全要求整段补全几百毫秒内完成。
- 离线任务的 SLO 是“今晚跑完”，只看成本。

负载画像决定优化方向

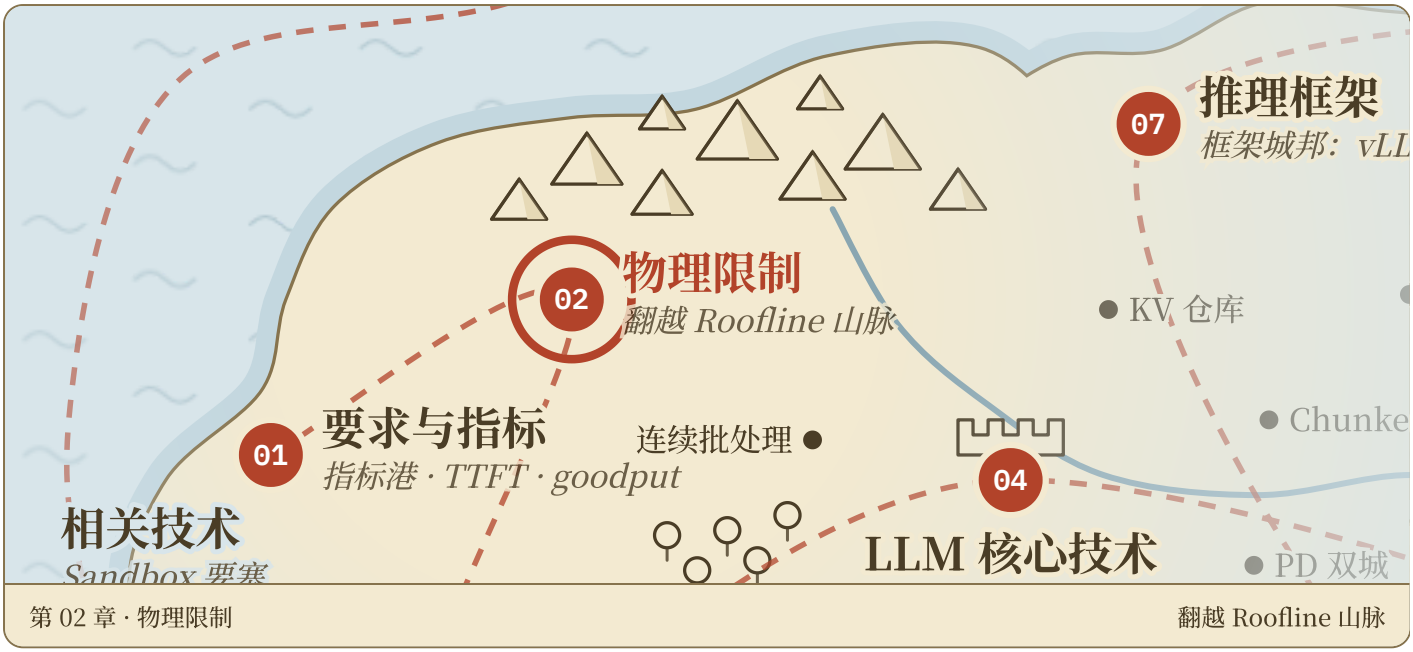
- **ISL / OSL（输入 / 输出长度）。**RAG、摘要、agent 是长输入短输出（prefill 重）；推理模型（o1 / R1 类）是短输入长输出（decode 重）。基准测试一定要说明长度分布。
- **前缀共享。**系统 prompt、few-shot、多轮对话历史、agent 的工具定义，都让大量请求共享前缀 → 前缀 cache 命中率可以很高（官方 API 对 cached input 打折就是这个原因）。
- **到达模式。**泊松到达 vs 突发；闭环（固定并发，发完一个再发下一个）vs 开环（按速率发）。闭环测试会掩盖排队问题。
- **多轮与会话。**同一会话的下一轮要能找到上一轮的 KV，这影响路由（会话粘性 / KV 感知路由）。

不同应用场景的推理负载画像

同样叫“推理”，不同场景的模型结构不同、用户需求不同，落到 GPU 上的负载形状就完全不同。下表是全文的索引：每一行在对应章节里都有“模型特征 → 场景 → 负载”的详细拆解。

场景	典型输入 / 输出	并发与批	延迟目标	跨步状态	主要瓶颈	关键技术
在线聊天	1–8k / 几百	高并发，大批	TTFT 秒内，TPOT 几十 ms	KV cache	decode 带宽、KV 容量	连续批、paged KV、前缀 cache、量化
推理模型 / RL rollout	短 / 数千到数万	极大批，离线	step 时间（看长尾）	KV + 每步更新的权重	长尾、权重同步、训推不一致	异步 RL、partial rollout、确定性 kernel
Coding agent	5–20 万（逐轮增长）/ 每轮数百到数千	中等并发，长会话	每轮 TTFT、tokens/s/user	跨工具调用保留的 KV	长 prefill、KV 保留与命中率	前缀 cache、KV 分层、会话粘性路由、sandbox
超大 MoE	同聊天	需要极大全局批	同聊天	KV（常为 MLA）	all-to-all、专家不均、小 GEMM	宽 EP、DP attention、DeepEP、EPLB、TBO

场景	典型输入 / 输出	并发与批	延迟目标	跨步状态	主要瓶颈	关键技术
Omni / 语音	音频 / 图像 + 文本 / 流式音频	中等，按阶段不同	首包几百 ms，RTF < 1	多阶段 KV + 会话状态	阶段间流水、小模型高频 decode	阶段化运行时、chunk 交接、CUDA graph
图像 / 视频扩散	prompt / 数千到数万 latent token × N 步	小批	秒到分钟	无 KV（步间特征相似）	算力、超长序列 attention	蒸馏、特征缓存、序列并行、稀疏 attention
机器人	多相机 + 指令 / 动作块	batch = 1（每台）	控制周期内，看最坏情况	动作缓冲区	prefill 延迟、抖动、边缘算力	action chunk、异步 + RTC、快慢双系统
AI for Science	序列 / MSA / 原子图 / 气象场	batch = 1 或天然大批	多为离线作业	成对表示、MD 状态	N^2 显存、CPU 预处理、精度	分块、三角 / 等变 kernel、MSA 缓存



02 物理限制：为什么 LLM 推理难

所有优化技术都是在和下面几条硬约束周旋。面试里能把这几条讲清楚并给出数量级，比背技术名词有用得多。

限制 1：自回归，token 只能一个一个生成

第 t+1 个 token 依赖第 t 个 token 的采样结果，所以 decode 无法在时间维度并行。一个 1000 token 的回答至少要 1000 次串行前向。并行只能来自三处：同时服务多个请求（批）、一次猜多个 token 再验证（投机解码）、或者换模型（多 token 预测、扩散 LM）。

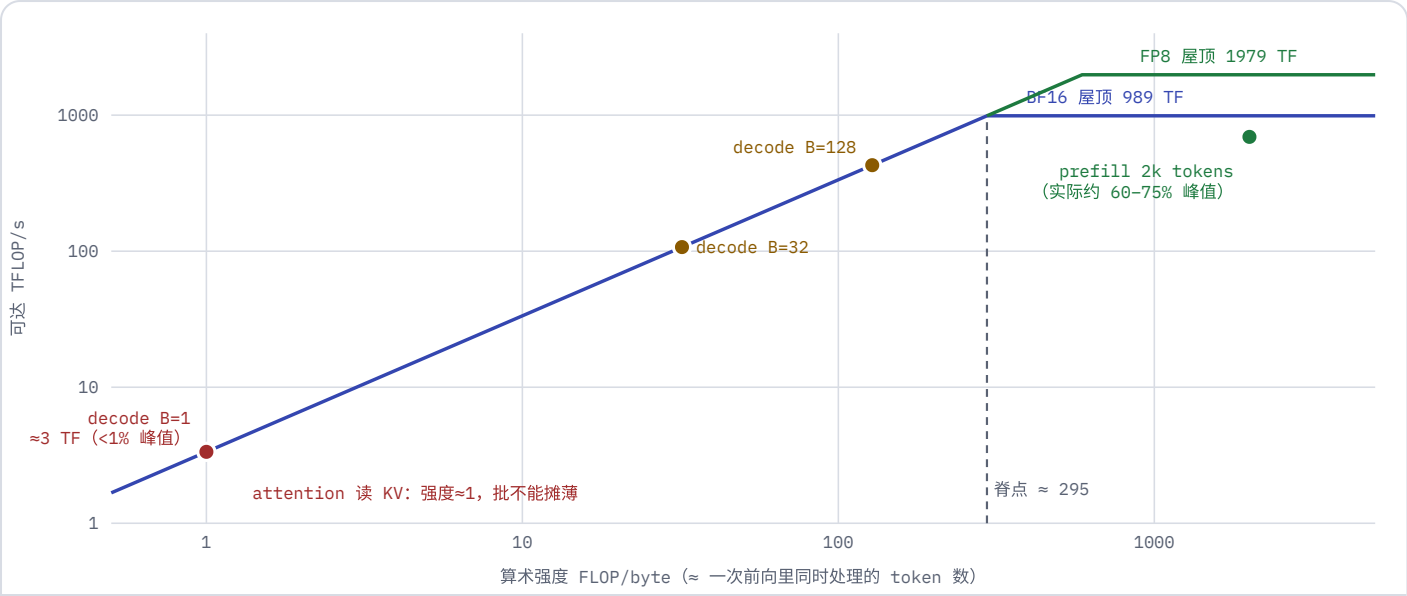
限制 2：prefill 吃算力，decode 吃带宽

一次前向对每个参数大约做 2 次浮点运算（乘 + 加）。设权重 BF16（每参数 2 字节），处理 `n` 个 token：

```
FLOPs ≈ 2 × N_params × n          读权重字节 ≈ 2 × N_params
算术强度 (arithmetic intensity) ≈ FLOPs / Bytes ≈ n      # 单位 FLOP/byte，只算权重

H100 SXM: BF16 稠密峰值 ≈ 989 TFLOP/s，HBM3 带宽 ≈ 3.35 TB/s
脊点 (ridge point) ≈ 989e12 / 3.35e12 ≈ 295 FLOP/byte
→ 一次前向里同时处理的 token 数 n < ~300 时受带宽限制，> ~300 时受算力限制
```

- **Prefill**：一个 2000 token 的 prompt 一次前向就处理 2000 个 token，强度远超脊点，是 **compute-bound**。
- **Decode**：每个请求每步只处理 1 个 token。批大小 B 的 decode 对权重的强度约为 B，B 不到几百就是 **memory-bound**。更糟的是 attention 部分要读每个请求自己的 KV cache，这部分**不能**被批摊薄，强度始终约为 1。
- 推论：decode 的 batch=1 速度上限 ≈ 带宽 / 权重字节。8B 模型 BF16 约 16 GB，H100 上每步 ≥ 4.8 ms，即单流最多约 200 token/s（还没算 KV）。



H100 roofline（对数坐标，示意）。斜线是带宽屋顶（3.35 TB/s × 强度），平线是算力屋顶。decode 的点在斜线上：加大批会沿斜线往右上走，直到碰到脊点；attention 读 KV 的部分一直卡在左下角。FP8 抬高算力屋顶，同时让每参数只占 1 字节、等效带宽翻倍。

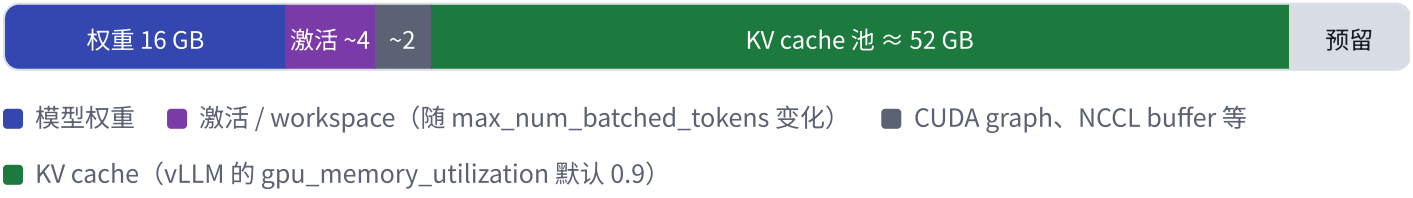
限制 3：KV cache 吃显存，而且随并发和上下文线性增长

为了不重复计算历史 token，每层的 Key / Value 要缓存下来。每个 token 的 KV 大小：

```
KV bytes / token = 2（K和V） × n_layers × n_kv_heads × head_dim × bytes_per_elem
总 KV = 上式 × Σ 每个并发请求的（输入 + 已生成）token 数
```

模型	结构	每 TOKEN KV（BF16）	一个 32K 上下文的请求
Llama-3-8B	32 层，GQA 8 个 KV head，head_dim 128	2×32×8×128×2 = 128 KiB	4 GiB
Llama-3-70B	80 层，GQA 8 KV head，head_dim 128	320 KiB	10 GiB
Qwen3-30B-A3B	48 层，GQA 4 KV head，head_dim 128	96 KiB	3 GiB
DeepSeek-V3	61 层，MLA：每层只存 512 维压缩 latent + 64 维 RoPE key	576×61×2 ≈ 69 KiB	≈ 2.1 GiB（671B 模型却比 8B 的 Llama 还小）

一块 80 GB 的 H100 跑 Llama-3-8B（BF16）的显存账大致如下。KV 池决定了最大并发：约 52 GB ÷ 128 KiB ≈ 40 万 token，平均每请求 4k token 时约 100 个并发。并发上不去，批就大不了，decode 就一直在带宽屋顶的低处。



限制 4：互联带宽比显存带宽低一到两个数量级

模型放不进一张卡就要切分，而切分后的通信走的链路越来越慢。设计并行策略的原则是：通信最频繁的切分（TP）放在最快的链路（NVLink）里。

链路	量级（单 GPU，单向约值）	用来做什么
HBM（片上显存）	3.35 TB/s（H100）→ 8 TB/s（B200）	读权重、读 KV
NVLink（机内）	H100 NVLink4 约 450 GB/s 单向（900 GB/s 双向）；Blackwell NVLink5 翻倍	TP 的 all-reduce、机内 EP
PCIe Gen5 x16	约 64 GB/s	CPU↔GPU、KV offload 到内存
InfiniBand / RoCE	400 Gb/s ≈ 50 GB/s 每网卡（通常 1 GPU 配 1 网卡）	跨机 PP / EP、PD 分离的 KV 传输
NVL72 机柜	72 块 GPU 在一个 NVLink 域内	让超大 MoE 的 EP 也走 NVLink

限制 5：负载不可预测

- **输出长度事先未知**：调度器不知道一个请求会占多久显存，所以要么保守预留（浪费），要么允许超卖再抢占（复杂）。
- **请求异构**：10 个 token 的问候和 100k token 的代码库分析混在一个批里。
- **流量突发**：GPU 扩容是分钟级（拉镜像、加载几百 GB 权重），流量变化是秒级。

限制 6：其他常被忽视的瓶颈

- **CPU 开销**：Python 调度、构造输入张量、kernel launch。decode 一步 GPU 只要几毫秒，CPU 慢 1 ms 就是十几个百分点的损失。这是 CUDA graph、多进程架构、调度与计算重叠的动机。
- **尾延迟**：一个长 prompt 进来，整个批的 ITL 都受影响。
- **数值非确定性**：同一请求在不同批大小下结果可能不同（浮点加法不满足结合律、kernel 按批大小选不同切分）。RL 训练和评测对此很敏感，见 08。
- **功耗与成本**：大规模部署时电力和机柜才是上限，单位是“每瓦 token 数”。

主流加速器参数（厂商公开峰值，稠密，约值）

加速器	显存	带宽	BF16 / FP8 / FP4 稠密算力	备注
NVIDIA H100 SXM	80 GB HBM3	3.35 TB/s	989 TF / 1979 TF / —	Hopper，FP8 Transformer Engine
NVIDIA H200	141 GB HBM3e	4.8 TB/s	同 H100	更大显存，长上下文 / 大 MoE 受益
NVIDIA B200	约 180–192 GB HBM3e	约 8 TB/s	约 2.2 PF / 4.5 PF / 9 PF	Blackwell，原生 FP4（NVFP4）

加速器	显存	带宽	BF16 / FP8 / FP4 稠密算力	备注
GB200 / GB300 NVL72	72 GPU 一个 NVLink 域	NVLink5 1.8 TB/s 双向每 GPU	—	大规模 EP 与 PD 分离的主力形态
AMD MI300X	192 GB HBM3	5.3 TB/s	约 1.3 PF / 2.6 PF / —	ROCm；大显存适合单卡放大模型
AMD MI355X	288 GB HBM3E	8 TB/s	支持 FP6 / FP4	CDNA4
Google TPU v6e (Trillium)	32 GB HBM	约 1.6 TB/s	约 918 TF BF16	JAX / vLLM TPU / SGLang TPU 后端

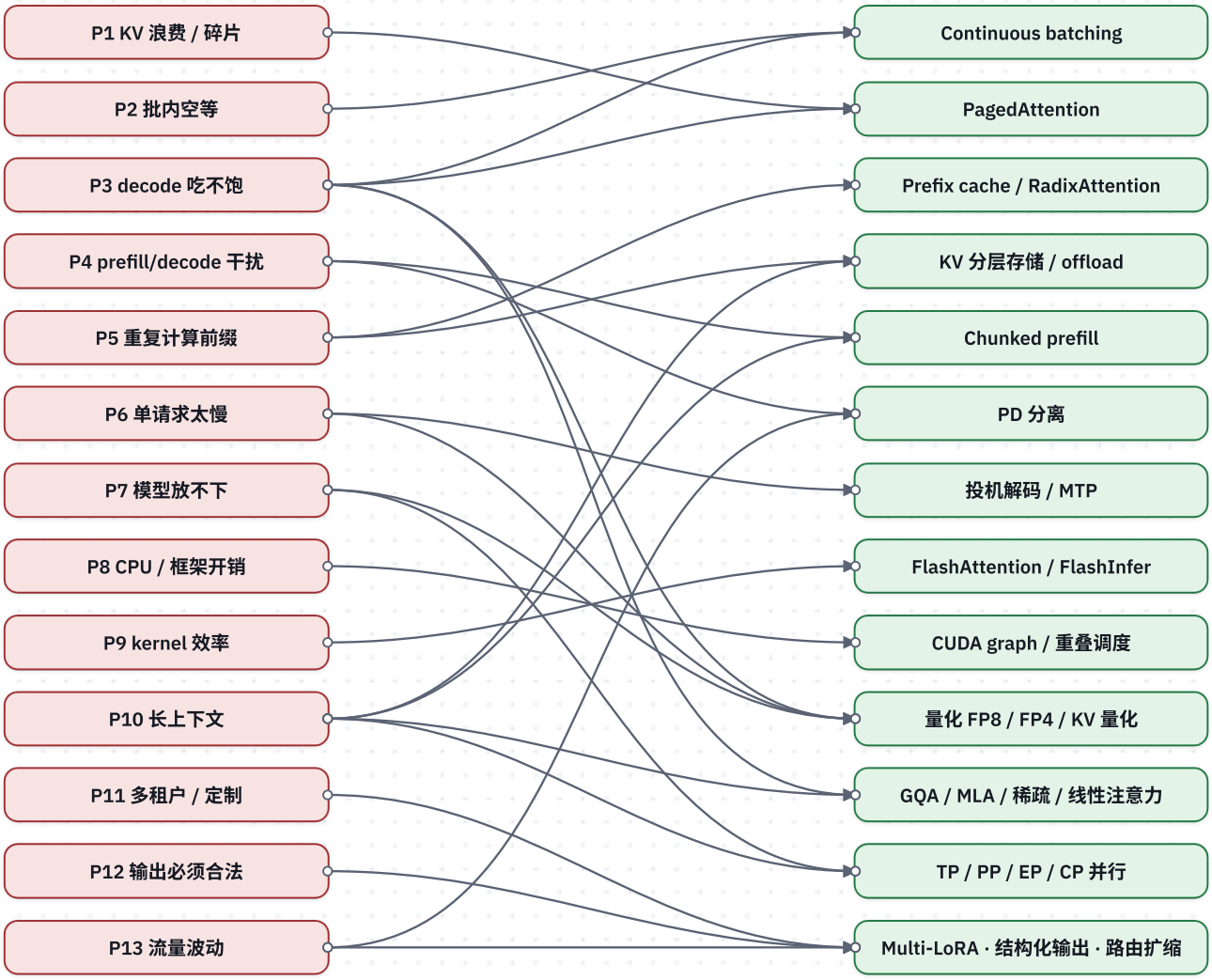


03 需要解决的问题

把上一节的限制落到一个朴素实现上（HuggingFace `generate()` 套一层 HTTP）会暴露出下面这些具体问题。后面每一项技术都可以对应回这张表；面试时先说“它解决哪个问题”，再说怎么做。

#	问题	朴素实现里的表现	根因（对应限制）
P1	KV 显存浪费与碎片	按 max_len 给每个请求预留连续 KV 空间，实际利用率只有 20–40%（vLLM 论文的测量）	输出长度未知 + 连续分配
P2	批内空等	静态批里短请求结束后，GPU 槽位一直空到最长的请求结束	请求异构
P3	decode 吃不饱 GPU	并发小 → 批小 → 带宽利用率低、每 token 成本高	decode 是 memory-bound，且 KV 占满显存
P4	prefill 与 decode 互相干扰	一个长 prompt 进来，所有正在生成的请求卡住几百毫秒	两阶段算力特征不同却共享 GPU
P5	重复计算	同一系统 prompt / 多轮历史每次都重新 prefill	KV 没有跨请求复用
P6	单请求太慢	小批时每 token 时间由读权重决定，再优化 kernel 也有下限	自回归串行
P7	模型放不下 / 太慢	70B BF16 要 140 GB；671B 要上千 GB	显存容量、互联带宽
P8	CPU 与框架开销	Python 调度、kernel launch 占 decode step 的显著比例	decode step 只有几毫秒
P9	kernel 效率低	attention 物化 N×N 矩阵；小 GEMM 用不满 Tensor Core	访存模式、硬件特性
P10	长上下文	128k 上下文时 KV 和 attention 计算爆炸	KV 线性增长、attention 二次方
P11	多租户与定制	每个客户一个微调模型，每个都要单独部署	权重不共享
P12	输出必须合法	要求 JSON / 函数调用，模型偶尔输出非法格式	采样是逐 token 的

#	问题	朴素实现里的表现	根因（对应限制）
P13	流量波动与成本	按峰值部署浪费，按均值部署会排队	扩容慢、负载突发



问题 → 技术映射。悬停左侧某个问题，可以只看解决它的技术；悬停右侧某项技术，可以看它同时缓解了哪些问题（例如 PagedAttention 同时缓解 P1 和 P3）。



04 LLM serving 的核心技术

这一节按“它解决哪个问题”组织。现代引擎（vLLM、SGLang、TensorRT-LLM）几乎都实现了全部这些技术，差异在实现细节、默认值和组合方式上。

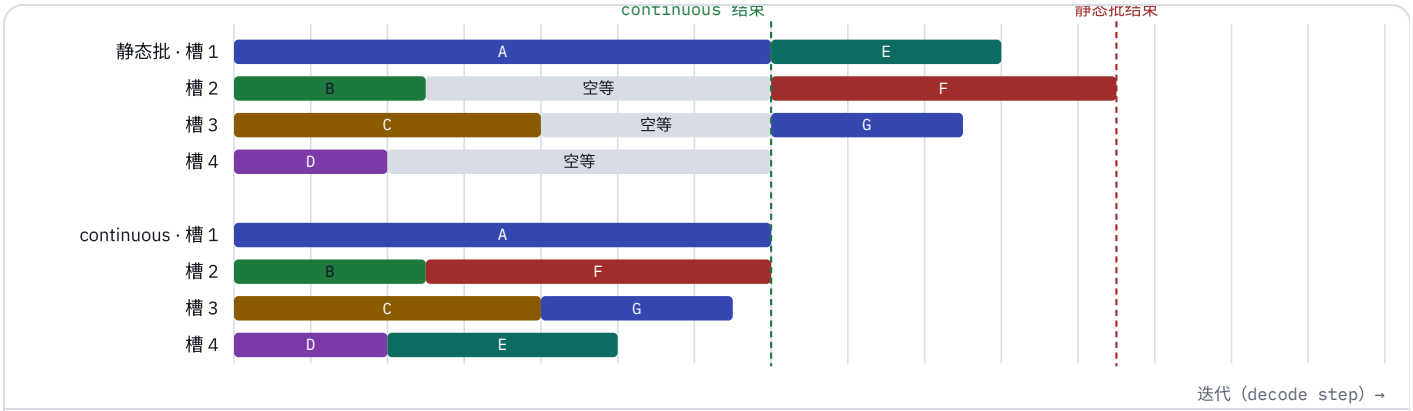
通用 LLM：模型特征 → 场景 → 推理负载

模型特征	为什么这样设计（场景决定）	在推理负载上的体现
Decoder-only Transformer，自回归逐 token 生成	训练目标是 next-token prediction，生成即逐步采样	decode 串行、每步读全部权重 → memory-bound；并行只能来自批、投机或多 token 预测
KV cache（因果注意力，历史 K/V 不变）	避免每步重算历史 token	显存随并发 × 上下文线性增长，成为并发上限 → 分页、前缀复用、分层存储、KV 量化
GQA / MLA 等压缩 KV 的注意力	长上下文和高并发是主流需求	每 token KV 小一个数量级 → 同样显存下批更大；但需要专门 kernel 与并行方式（DP attention）
超大词表（15 万+）与长上下文（128k+）	多语言、代码、长文档	LM head GEMM 与采样不再可忽略；长 prompt 的 prefill 成为 TTFT 主体
聊天模板 + 系统 prompt + 工具定义	产品形态是多轮对话与 agent	大量请求共享前缀 → 前缀 cache 命中率高；会话粘性路由

4.1 批处理与调度：continuous batching

源码 `vLLM scheduler.py` `vLLM engine/core.py` `persistent batch` `SGLang scheduler.py` `schedule_policy.py` `schedule_batch.py` `TRT-LLM py_executor`

解决 P2、P3。朴素的静态批（static batching）把 N 个请求凑成一批，直到最长的那个结束才能换下一批。Orca（OSDI’22）提出 **iteration-level scheduling**：调度粒度从“一个请求”降到“一次前向迭代”，每一步结束后，完成的请求立刻退出、等待的请求立刻补进来。这就是 continuous batching（也叫 in-flight batching）。



同样 7 个请求、4 个槽位。上：静态批，灰色是空等的槽位，E/F/G 要等第一批全部结束。下：continuous batching，每次迭代后空槽立刻被新请求填上，总时间短得多。

和 continuous batching 配套的几件事

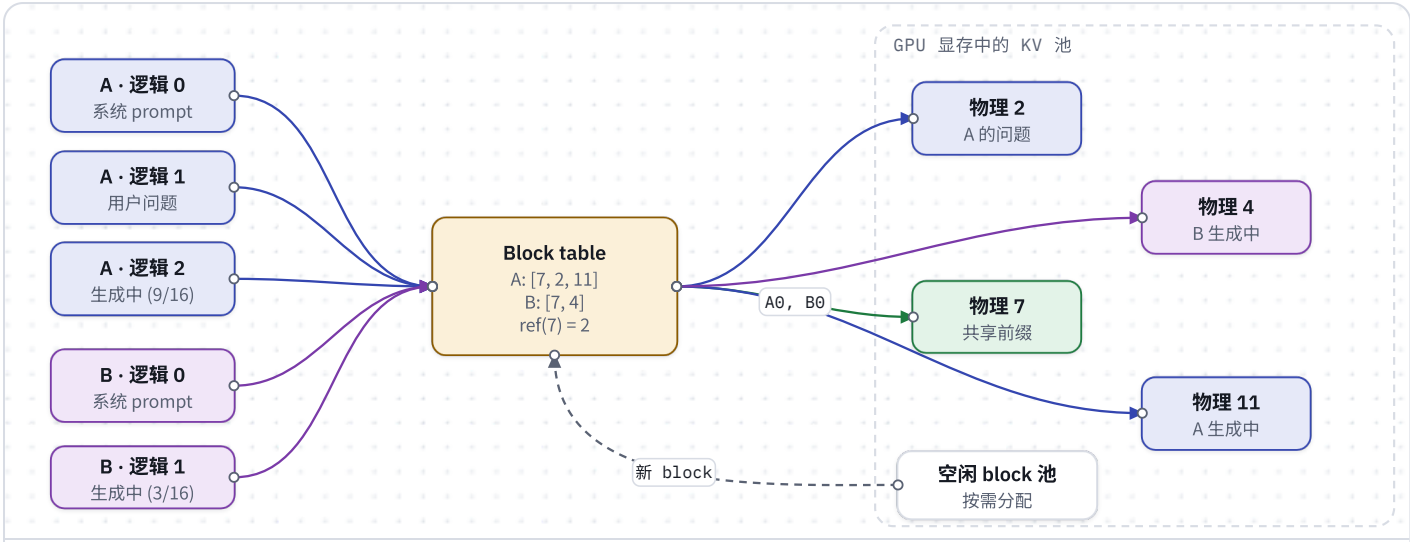
- **Selective batching / 扁平化。**线性层（QKV 投影、MLP）不关心 token 属于哪个请求，把所有请求的 token 拼成一个 `[总 token 数, hidden]` 张量一起算，没有 padding；只有 attention 按请求分开（varlen / paged attention kernel）。
- **Token budget。**每步最多处理 `max_num_batched_tokens` 个 token、最多 `max_num_seqs` 个请求，控制单步时长和激活显存。vLLM V1 把 prefill 和 decode 统一看成“给每个请求分配本步要算的 token 数”，于是 chunked prefill、前缀 cache、投机解码都落在同一套逻辑里。
- **抢占（preemption）。**KV 不够时把某些请求踢出：*recompute*（丢掉 KV，之后重新 prefill，vLLM 默认）或 *swap*（KV 拷到 CPU 内存）。抢占频繁说明并发超过了 KV 容量。
- **调度策略。**FCFS 最常见；还有优先级调度、公平调度（如 OSDI’24 的 VTC，按租户已服务 token 数做公平）、按预测输出长度排序（近似最短作业优先，降低平均延迟）、SLO 感知调度。SGLang 还有 cache-aware 调度：优先调度前缀命中长的请求。

4.2 KV cache 管理：PagedAttention、前缀缓存、分层存储

PagedAttention（vLLM，SOSP ’23）

源码 `kv_cache_manager.py` `block_pool.py` `single_type_kv_cache_manager.py` `cache_kernels.cu` `SGLang memory_pool.py`

解决 P1，间接解决 P3。借鉴操作系统的虚拟内存分页：KV 被切成固定大小的 block（默认 16 个 token），每个请求有一张 **block table** 把逻辑 block 映射到任意位置的物理 block，按需分配。浪费只剩每个请求最后一个没填满的 block。论文报告已有系统浪费 60–80% 的 KV 显存，vLLM 降到 4% 以下，吞吐提升 2–4 倍。分页还让共享变得容易：并行采样、beam search 的公共前缀只存一份，靠引用计数 + copy-on-write。



Block table。请求 A、B 的逻辑 block 连续，物理 block 可以散落在显存任意位置。两个请求共享系统 prompt 所在的物理 block 7（引用计数 = 2），写入时才复制。attention kernel 根据 block table 间接寻址。

前缀缓存：vLLM APC 与 SGLang RadixAttention

源码 `vLLM block 哈希 kv_cache_utils.py` `SGLang radix_cache.py`

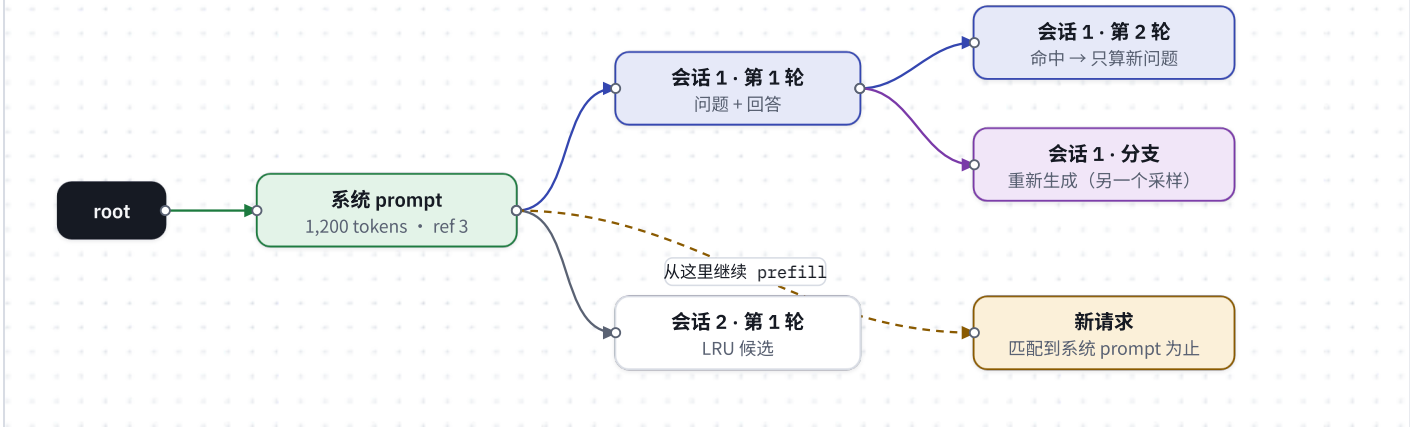
解决 P5。请求结束后不立刻释放 KV，而是留作 cache；新请求如果前缀相同就直接复用，只 prefill 不同的部分，TTFT 和算力都省下来。两种主流实现：

vLLM Automatic Prefix Caching

以满 block 为单位，每个 block 的 key = hash(父 block 的 hash, 本 block 的 token, 额外信息如 LoRA / 多模态 hash)。全局哈希表查找；引用计数为 0 的 block 进入 LRU 空闲队列，被复用前仍可命中。实现简单、开销 O(1)。

SGLang RadixAttention

把所有缓存的 token 序列组织成一棵 radix tree（压缩前缀树），边上是 token 序列，节点挂着对应的 KV。匹配就是从根往下走最长前缀；淘汰按 LRU 从叶子开始。配合 cache-aware 调度（优先最长前缀匹配），对多轮对话、few-shot、tree-of-thought 等复杂共享模式很友好。



Radix tree 示意。三个请求共享系统 prompt；其中两个是同一会话的多轮对话，共享到第一轮回答。新请求只需从匹配结束的节点继续 prefill。被引用的节点不能淘汰，叶子按 LRU 淘汰。

KV 分层存储与 offload

源码 `vLLM kv_offload/` `SGLang hicache_storage.py` `LMCache` `Mooncake Store` `Dynamo kv_router`

解决 P5、P10。GPU 显存装不下所有值得缓存的前缀（多轮对话、agent 的长上下文、多副本之间的共享），于是把 KV 下沉到 CPU 内存、本地 SSD、甚至远端分布式存储，命中时再载回。判断是否值得的数量级：

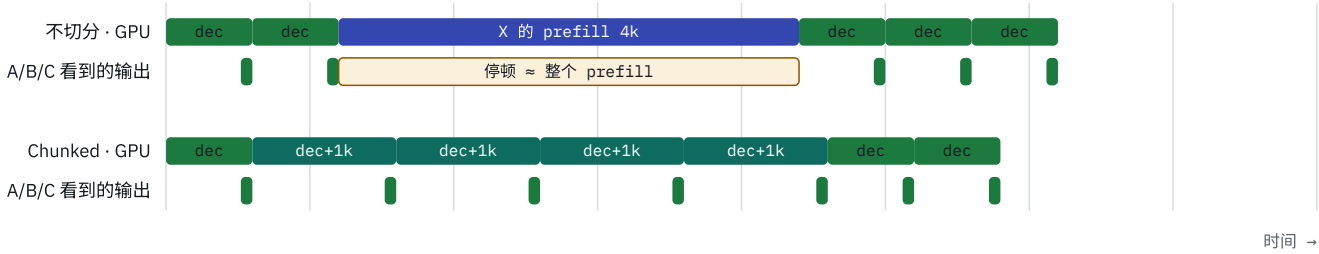
```
Llama-3-8B，10k token 前缀：KV = 10k × 128 KiB ≈ 1.25 GiB
  从 CPU 载回（PCIe Gen5，约 50 GB/s 实际）：≈ 25 ms
  重新 prefill：2 × 8e9 × 1e4 = 1.6e14 FLOP，按 600 TFLOP/s 有效算力 ≈ 270 ms
→ 载回比重算快约 10 倍；上下文越长、模型越大，差距越大
```

- **开源实现：**LMCache（vLLM 生态的 KV 存储层，CPU / 磁盘 / 远端后端）、Mooncake Store（Kimi 的 KVCache-centric 架构，利用集群空闲的 CPU 内存和 SSD，RDMA 传输）、SGLang HiCache（GPU → CPU → 存储三级 radix cache）、NVIDIA Dynamo KVBM（KV block manager）。
- **商业 API 的体现：**DeepSeek 的上下文硬盘缓存、Anthropic / OpenAI 的 prompt caching（命中部分价格大幅打折）。
- **KV 量化（FP8 / INT8）**也是一种“扩容”：容量翻倍、attention 读带宽减半，通常精度损失很小。

4.3 Chunked prefill：把 prefill 切碎塞进 decode 批

源码 `vLLM: token budget 内拆分 prefill` `SGLang: PrefillAdder 与 chunked req`

解决 P4。长 prompt 的 prefill 一次要跑几百毫秒，期间同一 GPU 上所有 decode 请求都停住，ITL 出现尖峰。Sarathi-Serve（OSDI’24）把 prefill 切成固定大小的 chunk，每次迭代 = 所有 decode token + 一个 prefill chunk，凑满 token budget。这样 decode 永不被长时间阻塞（stall-free），而 prefill chunk 正好把 decode 批“吃不饱”的算力用掉。



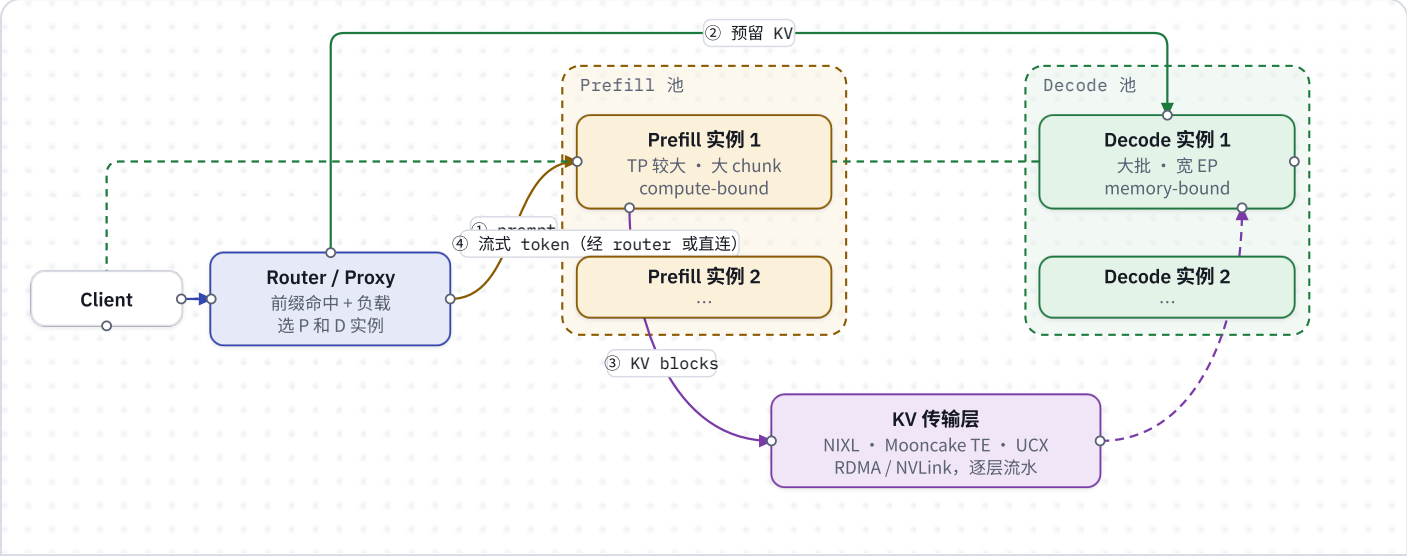
上：不切分。请求 X 的 4k prefill 独占一次迭代，A/B/C 的 decode 停顿。**下：chunked prefill。**每次迭代混合 decode + 1k 的 prefill chunk，单步变长一点点，但不再有停顿。

- **权衡：**chunk 越小，ITL 越平稳，但 prefill 效率下降（每个 chunk 要重读之前 chunk 的 KV、GEMM 变小），X 自己的 TTFT 变长。常见 token budget 是 2k–8k，根据 TPOT SLO 调。
- **与 PD 分离的关系：**chunked prefill 是“同一 GPU 内做时间上的隔离”，PD 分离是“换不同 GPU 做空间上的隔离”。中小规模部署用 chunked prefill 通常就够了；它在 vLLM V1 中默认开启。

4.4 Prefill–Decode 分离（PD disaggregation）

源码 `vLLM KVConnector 接口` `NixlConnector` `LMCacheConnector` `SGLang PD prefill.py` `decode.py` `mooncake/conn.py` `NIXL API` `Mooncake Transfer Engine`

解决 P4，并让两阶段可以独立选并行度、硬件和扩缩容。prefill 是 compute-bound、关心 TTFT；decode 是 memory-bound、关心 TPOT 和大批。放在一起时，无论怎么调度都在两个 SLO 之间妥协。DistServe（OSDI’24）、Splitwise（ISCA’24）、Mooncake（FAST’25）把它们放到不同的 GPU 池，prefill 算完后把 KV cache 通过 RDMA / NVLink 传给 decode 实例。DistServe 报告在同样 SLO 下最多可服务 7.4 倍的请求。



PD 分离的数据流。路由器选择 prefill 和 decode 实例（考虑前缀命中和负载）；decode 端先预留 KV 空间；prefill 逐层算完后把 KV 通过 RDMA 直接写进 decode 端显存（可与计算重叠）；之后 decode 端独立生成，流式返回。

收益

- TTFT 与 TPOT 解耦，各自满足 SLO，goodput 提升。
- 两池可以用不同并行度：prefill 用 TP 降 TTFT，decode 用大 DP + 宽 EP 撑大批（DeepSeek 就是这样，见 05）。
- P:D 比例按负载独立扩缩（长 prompt 多就多加 P）。
- 可以用不同硬件（如算力强的卡做 P，显存带宽大的卡做 D）。

代价与适用条件

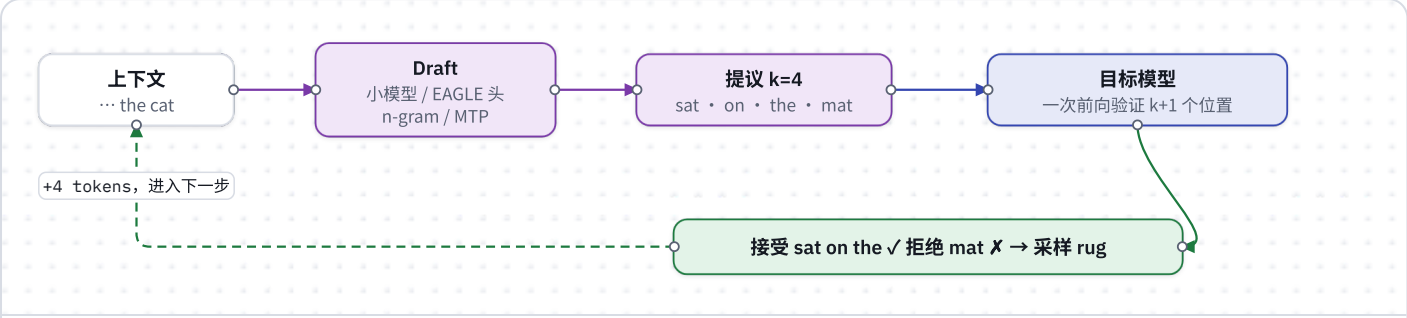
- 要传 KV：70B 模型 4k prompt 约 1.25 GiB；单张 400G 网卡约 25 ms，8 卡并行约 3 ms。需要 RDMA 和逐层流水才能藏住。
- 系统复杂：两套实例、路由、失败处理、KV 生命周期。
- 小模型、短 prompt、小集群时收益有限，chunked prefill 更划算。

- **实现：**vLLM 的 KV Connector 抽象（NixlConnector、LMCacheConnector、P2pNcclConnector、Mooncake 等），SGLang 的 PD 模式（Mooncake / NIXL 传输后端 + bootstrap 握手），NVIDIA Dynamo 和 llm-d 在集群层编排 PD。
- **进一步的拆分：**EPD（把多模态 encoder 也拆出去，见 09）、AF 分离（attention 与 FFN/专家分到不同 GPU，例如字节的 MegaScale-Infer、阶跃星辰 Step-3 的 AFD），本质都是“按算力特征把模型的不同部分放到最合适的资源上”。

4.5 投机解码（Speculative decoding）

源码 `vLLM eagle.py` `ngram_proposer.py` `rejection_sampler.py` `SGLang eagle_worker_v2.py` `EAGLE 官方` `Medusa`

解决 P6。decode 小批时 GPU 算力大量闲置，验证 k 个 token 的成本和生成 1 个几乎一样（都是读一遍权重）。于是：用便宜的方法先猜 k 个 token（draft），再让目标模型一次前向并行验证，接受最长的正确前缀，外加目标模型在第一个拒绝位置自己采样出的一个 token。用拒绝采样（rejection sampling）做接受判断时，输出分布与目标模型**完全一致**（Leviathan et al. 2022；Chen et al. 2023）。



一次投机步。draft 提议 4 个 token，目标模型一次前向验证；前 3 个被接受，第 4 个被拒绝，目标模型在该位置重新采样，共前进 4 个 token，只花了 1 次目标前向（外加 draft 开销）。

```
每次验证期望前进的 token 数 = (1 − α^(k+1)) / (1 − α)        # α = 每个 draft token 的接受
例：α = 0.8, k = 4 → (1 − 0.8⁵)/0.2 ≈ 3.36 个 token / 次目标前向
实际加速 = 期望 token 数 ÷ (1 + draft 开销占比 + 验证比单 token 多出的开销)
```

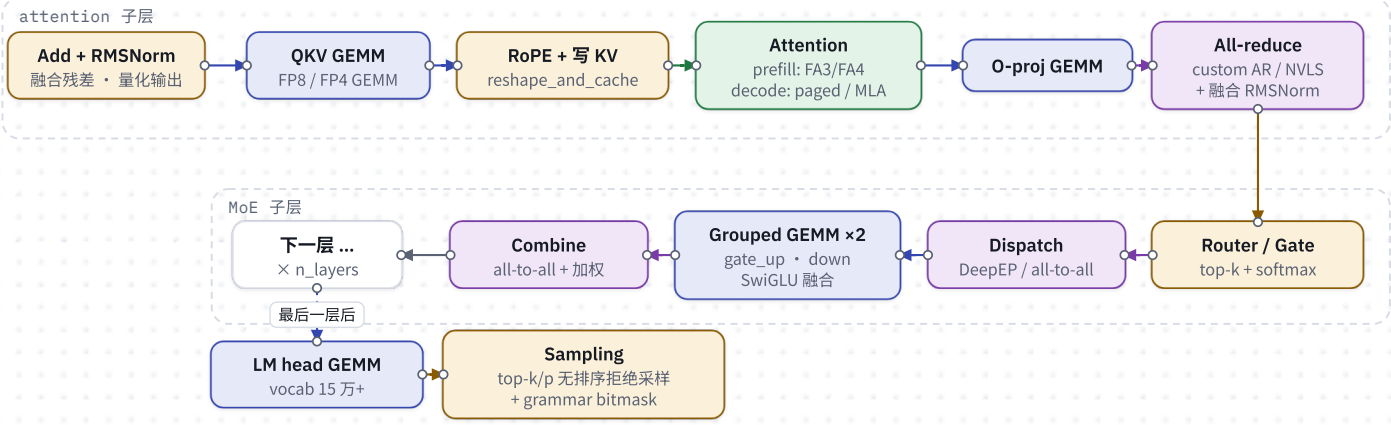
方法	DRAFT 从哪来	特点
独立 draft 模型	同 tokenizer 的小模型（如 1B 给 70B 打草稿）	通用；但要额外显存和调度，小模型与大模型分布差距决定 α
n-gram / prompt lookup	从 prompt 或已生成文本里找匹配的后续	零训练、零显存；代码编辑、RAG、改写等“输出大量复制输入”的场景很有效
Medusa	在目标模型最后一层上加多个预测头，各预测 +1、+2··· 位置	需要训练头；树状候选 + tree attention 验证
EAGLE / EAGLE-2 / EAGLE-3	轻量的自回归头，输入目标模型的隐藏特征（EAGLE-3 融合多层特征）	接受率高，SGLang / vLLM 主力方案；EAGLE-2 动态草稿树
MTP（Multi-Token Prediction）	模型训练时自带的多 token 预测模块（DeepSeek-V3、部分 Qwen / GLM / MiMo 模型）	无需另训 draft；DeepSeek-V3 报告第 2 个 token 接受率约 85–90%
Lookahead / Jacobi	目标模型自己并行迭代猜测	无 draft，但收益通常较小

什么时候投机解码反而变慢

大批量时 GPU 已接近 compute-bound，多验证的 k 个 token 不再“免费”，被拒绝的 token 是纯浪费。高温采样、开放式创作时 α 低。因此生产系统常按负载动态调 k 或关闭投机；面试时要能说出“投机解码用算力换延迟，适合低并发、延迟敏感的场景”。

4.6 Kernel 与 runtime：把每一步做快

解决 P8、P9。批处理和调度决定“算什么”，kernel 和 runtime 决定“算得多快”。理解 kernel 最好的办法是沿着一层 decoder 的执行顺序，看每一步调用了哪个 kernel、它是 compute-bound 还是 memory-bound、和谁可以融合。



一层 decoder（MoE 模型）在 decode 时的 kernel 序列。蓝色是 GEMM 类（Tensor Core），绿色是 attention，紫色是通信，橙色是可以被融合掉的小 kernel。实线框内的一串小 kernel 在优化后通常融合为 1–2 个 launch；整层再被 CUDA graph 录制成一次提交。

按类别看 kernel：要解决什么、代表实现、源码

类别	核心难点	代表实现与要点	源码入口
Prefill attention	$O(N^2)$ 计算，不能物化 N×N 分数矩阵	FlashAttention：分块 + online softmax，Q 块常驻 SRAM，循环读 K/V 块。FA2 改并行划分（按序列分块到 thread block）；FA3（Hopper）用 TMA 异步搬运 + WGMMA + warp specialization（生产者 warp 搬数据、消费者 warp 算）+ ping-pong 调度隐藏 softmax；FA4（Blackwell）用 CuTe DSL 编写，利用 tcgen05 / TMEM	`FA2 kernel` `FA3 hopper/` `FA4 cute/`
Decode / paged attention	每个请求只 1 个 query，KV 散落在分页 block 里；批小时 SM 吃不满	按 block table 间接读 KV；split-KV（Flash-Decoding）沿序列切分再归约；FlashInfer 用统一的 block-sparse 格式 + 运行时 *plan* 做负载均衡；cascade attention 让共享前缀只读一次；TRT-LLM 的 XQA 针对 GQA 把同组 query head 合并成一个 tile	`decode.cuh` `prefill.cuh` `scheduler.cuh` `cascade.cuh` `page.cuh` `TRT-LLM XQA`
MLA attention	latent 维度 512 + RoPE 64，decode 时要“吸收”上投影	FlashMLA（Hopper / Blackwell，含稀疏版本供 DSA 使用）；FlashInfer MLA；推理时把 W_UK 吸收进 Q、W_UV 吸收进 O，attention 直接在 latent 上算（MQA 形式）	`FlashMLA 接口` `FlashMLA csrc` `vLLM MLA backends`

类别	核心难点	代表实现与要点	源码入口
线性注意力 / SSM	状态递推是串行的，要变成可并行的 chunk 形式	chunkwise 并行：块内用矩阵乘，块间传递状态；Mamba 的 selective scan、causal-conv1d；flash-linear-attention（fla）收录 GLA、DeltaNet、Gated DeltaNet 等 Triton kernel	`fla/ops` `mamba selective_scan` `vLLM mamba/GDN 层`
Dense GEMM	decode 时 M（token 数）很小，是“瘦”矩阵；量化后要带 scale	cuBLAS / cuBLASLt；CUTLASS 模板（tile 形状、stage 数、epilogue 融合）；DeepGEMM：FP8 细粒度（1×128 / 128×128）scale，JIT 编译，代码量很小适合精读；split-K / stream-K 解决 M 小时 tile 数不够的问题	`CuTe 教程` `DeepGEMM impls` `CUTLASS`
混合精度 GEMM	INT4 / FP4 权重 × BF16 激活，反量化不能拖慢	Marlin：权重预重排，反量化与 Tensor Core 计算流水重叠，batch 16–32 内接近 4× 理论加速；Machete（Hopper 版）；MoE 版 Marlin	`Marlin` `vLLM MoE Marlin`
MoE GEMM	每个专家的 token 数不同且很少，形状随路由变化	按专家排序 token（align / permute）后做 grouped GEMM；Triton fused_moe（vLLM / SGLang 的通用路径）；DeepGEMM 的 contiguous（prefill）与 masked（decode，形状固定便于 CUDA graph）布局；CUTLASS grouped GEMM；FlashInfer / TRT-LLM 的 fused MoE	`vLLM fused_moe.py` `modular_kernel.py` `FlashInfer fused_moe`
通信	TP all-reduce 每层两次、EP all-to-all 延迟敏感	NCCL；小消息用自定义 one-shot / two-shot all-reduce（直接读写对端显存，IPC handle）；NVLink SHARP（NVLS）多播；DeepEP（normal + low-latency，NVSHMEM / IBGDA）、pplx-kernels；MSCCL++	`vLLM custom AR` `SGLang allreduce` `DeepEP kernels` `pplx-kernels` `NCCL device` `MSCCL++`
小算子与融合	memory-bound，launch 多	RMSNorm + 残差、SiLU×Mul、RoPE、per-token FP8 量化、KV 写入（reshape_and_cache）；融合进 GEMM 的 prologue / epilogue 或彼此融合	`layernorm` `activation` `RoPE` `cache_kernels`
采样	15 万词表上做 top-k / top-p，排序很贵	FlashInfer 的无排序拒绝采样（反复抽样 + 阈值，期望几轮收敛）；Triton top-k/top-p；结构化输出的 bitmask 在采样前按位应用	`sampling.cuh` `vLLM topk_topp` `vLLM sampler`

类别	核心难点	代表实现与要点	源码入口
Runtime：图与编译	每步几百个 launch，CPU 赶不上 GPU	CUDA graph 按批大小预捕获；vLLM V1 piecewise graph（attention 在图外）+ FULL graph 模式；torch.compile 自定义 pass（量化融合、AR 融合）；SGLang 分 prefill / decode 两类 graph runner	`cudagraph_dispatcher` `compilation/backends` `SGLang decode graph`
确定性 kernel	归约顺序随批大小变化 → 结果随批变化	batch-invariant 的 matmul / RMSNorm / attention：固定 split 策略，不随批大小切换 kernel 配置	`vLLM batch_invariant` `SGLang batch_invariant_ops`
Megakernel	batch=1 时 kernel 边界和 launch 本身就是开销	把整层乃至整个前向编译成一个持久化 kernel，在 SM 内调度任务图（Mirage Persistent Kernel；Hazy Research 的 Llama megakernel）	`Mirage MPK` `ThunderKittens kernels`

面试常问的 kernel 概念

概念	一句话解释	为什么推理里重要
Tiling / 分块	把大矩阵切成能放进 shared memory / 寄存器的块，块内复用数据	决定算术强度；FlashAttention、GEMM 的核心
Online softmax	流式维护 running max 和 running sum，分块算 softmax 而不需要整行	attention 不物化 N×N 的前提；split-KV 归约也靠它（合并 LSE）
Warp specialization	同一 block 内不同 warp 分工（搬数据 / 计算 / 写回），用 barrier 流水	Hopper 之后高性能 kernel 的标准写法（FA3、DeepGEMM）
TMA / WGMMA / tcgen05	Hopper 的张量内存加速器做异步块搬运，WGMMA 是 warpgroup 级 MMA；Blackwell 的 tcgen05 MMA 把累加器放到专用的 Tensor Memory（TMEM）	不同代际 kernel 无法直接复用，这就是为什么每代 GPU 都有新版 FA / GEMM
Persistent kernel / stream-K / split-K	常驻 kernel 循环领取 tile；沿 K 维切分以增加并行度	decode 的瘦 GEMM tile 太少，不切就吃不满 SM
Epilogue 融合	GEMM 写回前顺便做 bias、激活、量化、残差	省掉一次 HBM 往返
Occupancy 与寄存器压力	每个 SM 同时驻留的 warp 数受寄存器 / shared memory 限制	memory-bound kernel 需要足够多的在途访存来掩盖延迟
PDL / CUDA graph	Programmatic Dependent Launch 让下一个 kernel 提前启动 prologue；graph 一次提交整串 kernel	decode 一步只有几毫秒，launch 间隙占比显著

写 kernel 的工具链：CUDA C++ + CUTLASS / CuTe 仍是性能天花板；Triton 是最常用的 Python DSL（vLLM / SGLang 大量 MoE、attention 变体用它）；CuTe DSL（Python 版 CUTLASS，FA4

用它）、TileLang（DeepSeek 的稀疏 attention 等用它写参考实现）、ThunderKittens 让写 Hopper / Blackwell 专用 kernel 更容易。

工具链源码 `Triton` `TileLang examples` `ThunderKittens` `◈ FlashInfer · Code Wiki` `◈ FlashAttention · Code Wiki` `◈ DeepGEMM · Code Wiki`

4.7 量化：少读字节、多放 KV

源码 `vLLM fp8.py` `quantization/ 各方法` `llm-compressor` `Marlin`

解决 P3、P6、P7。decode 是 memory-bound，权重字节数减半，单步时间就接近减半；同时省下的显存给 KV，批更大。prefill 是 compute-bound，只有激活也量化（W8A8、FP8、FP4）且用对应的 Tensor Core 指令时才加速。

方案	量化对象	收益	代表方法 / 硬件	注意
Weight-only INT4 / INT8	只量化权重，计算时反量化	decode 加速明显、显存省 2–4×	GPTQ（二阶误差补偿）、AWQ（按激活幅度保护重要通道）；Marlin kernel	大批 / prefill 收益小，反量化有开销
W8A8 INT8	权重 + 激活	prefill 也加速	SmoothQuant（把激活离群值“平移”到权重）	激活离群值是难点
FP8（E4M3）	权重 + 激活，per-tensor / per-channel / 分块 scale	近乎无损，算力 2×	Hopper 起原生；DeepSeek-V3 用 128×128 分块 scale	当前生产最常用
FP4（NVFP4 / MXFP4）	权重（+ 激活）	再翻倍	NVFP4：16 元素一组、FP8 scale（Blackwell）；MXFP4：32 元素一组、E8M0 scale（OCP 标准，gpt-oss 原生发布）	更依赖 QAT / 好的校准
KV cache 量化	K / V	KV 容量 2×、attention 带宽减半	FP8 KV 最常见，也有 INT4 KV	长上下文任务要专门评测

工具：vLLM 的 `llm-compressor`、NVIDIA ModelOpt、Intel AutoRound、PyTorch torchao、bitsandbytes、llama.cpp 的 GGUF k-quants。面试常问“量化后怎么验证”：除了 perplexity，一定要在目标任务（数学、代码、长上下文检索）上评测，量化对推理链长的任务更敏感。

4.8 模型结构侧的优化：从源头减小 KV 与计算

源码 `vLLM MLA backends` `vLLM mamba / GDN` `按层类型的 KV 管理` `flash-linear-attention` `FlashMLA`

解决 P3、P10。很多“推理优化”其实发生在模型设计阶段，serving 系统要跟着支持。

结构	做法	KV / 计算影响	SERVING 要配合什么
MQA / GQA	多个 query head 共享一组 K/V head	KV 缩小 n_heads / n_kv_heads 倍（Llama-3-70B：64 / 8 = 8×）	TP 切分时 KV head 数可能小于 TP 数，需要复制 KV head
MLA（DeepSeek-V2/V3）	把 K/V 压成低秩 latent（512 维）+ 解耦的 RoPE key	KV 比同规模 MHA 小一个数量级以上	decode 时把上投影矩阵“吸收”进 Q 和 O；专用 kernel（FlashMLA、FlashInfer MLA）；TP 不切 latent → 常用 DP attention
滑动窗口 / 局部注意力	只看最近 W 个 token；常与全局层交错（Gemma 3 为 5:1）	局部层 KV 恒定	按层类型分开管理 KV（vLLM V1 hybrid KV allocator）
线性注意力 / SSM 混合	Mamba、Gated DeltaNet、Lightning Attention 等与少量全注意力层混合（Jamba、Qwen3-Next、MiniMax-M1、Kimi Linear）	大部分层只存固定大小的状态	状态不能按 token 切页，前缀 cache 需要定期存状态快照
稀疏注意力	每个 query 只取 top-k 相关 token（NSA、DeepSeek-V3.2 的 DSA）	长上下文 attention 计算接近线性	索引器 kernel + 稀疏 gather
MoE	每个 token 只激活少数专家	计算按激活参数算，显存按总参数算	见 05

4.9 并行策略：模型放不下、或一张卡太慢

源码 `vLLM parallel_state.py` `all2all.py` `SGLang DP controller` `Megatron tensor_parallel`

解决 P7、P10。推理里的并行与训练相似，但目标不同：训练追求吞吐，推理还要压单请求延迟，所以 PP 在推理中没那么受欢迎（它不降低单 token 延迟）。



五种切法。TP 切每层的矩阵（每层 2 次 all-reduce，必须在 NVLink 域内）；PP 切层（只在相邻 stage 间传激活，可跨机）；DP 复制整个模型；EP 把 MoE 专家分散（all-to-all）；CP/SP 沿序列维度切长 prompt（ring attention / Ulysses）。

并行	通信	优点	缺点	推理中的典型用法
TP	每层 2 次 all-reduce，量 ∝ batch_tokens × hidden	降低单 token 延迟，权重和 KV 都被切分	通信频繁，跨机代价大；TP 太大时 GEMM 变瘦	机内 TP=2/4/8
PP	相邻 stage 点对点传激活	通信少，可跨机	不降单 token 延迟；需要多个微批才能填满流水	超大模型跨机；吞吐优先
DP	无（路由层面分流）	线性扩吞吐，最简单	每个副本都要装下整个模型	多副本 + 负载均衡；MoE 的 attention 部分做 DP
EP	每个 MoE 层两次 all-to-all（dispatch / combine）	专家显存被分摊，每卡专家少 → 每专家批更大	负载不均、all-to-all 延迟	DeepSeek / Qwen MoE 的大规模部署
CP / SP	ring 传 KV 或 all-to-all 换 head	超长 prompt 的 prefill 可以多卡并行	decode 收益小	百万 token 级 prefill

4.10 结构化输出、多 LoRA、API 层功能

源码 `XGrammar cpp/` `vLLM xgrammar backend` `SGLang xgrammar backend` `vLLM tool_parsers/` `SGLang function_call/` `vLLM punica_wrapper` `SGLang lora_manager.py`

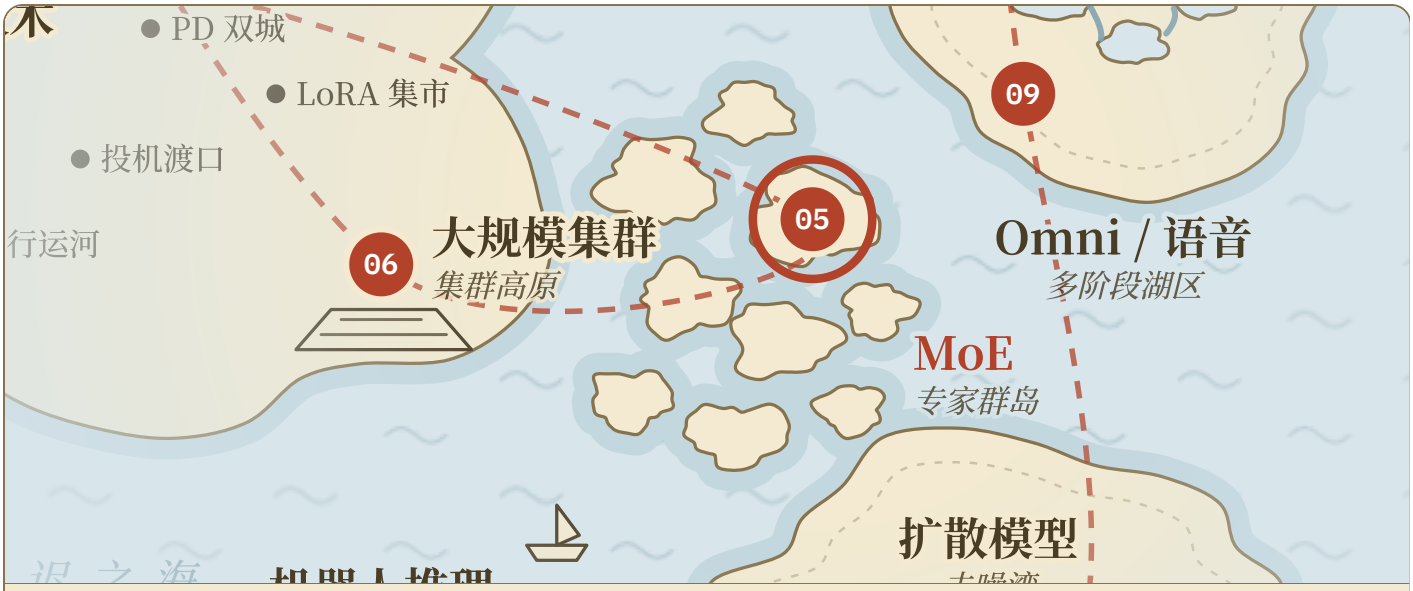
解决 P11、P12。这些不提升“裸性能”，但是产品必需品，而且做不好会严重拖慢吞吐。

- **结构化输出（constrained decoding）。**把 JSON schema / 正则 / CFG 编译成自动机，每步生成一个 token mask，把非法 token 的 logit 置为 −∞。难点是 mask 计算要在 GPU 前向期间完成、不成为瓶颈。XGrammar 用下推自动机 + 预计算“与上下文无关”的 token 集合，把大部分检查挪到预处理；llguidance、Outlines 是其他实现；SGLang 早期的 compressed FSM 可以一次“跳过”确定的多个 token（jump-forward）。
- **工具调用与推理内容解析。**不同模型的 tool call 格式各异，服务端需要 tool-call parser 和 reasoning parser 把输出转成 OpenAI 格式的 `tool_calls` 和 `reasoning_content`。
- **Multi-LoRA。**一个基座 + 成百上千个 LoRA adapter 同时服务。S-LoRA 把 adapter 权重与 KV 统一分页管理并按需从 CPU 载入；Punica 的 SGMV kernel 让一个批里不同请求用不同 adapter 仍然高效。vLLM / SGLang 都支持动态加载与批内混合 adapter。
- **多租户隔离。**优先级、配额、按租户的公平调度、按 token 计费与限流（TPM / RPM）。
- **OpenAI 兼容 API。**`/v1/chat/completions`、`/v1/completions`、`/v1/embeddings`、`/v1/audio/*`、Responses API；logprobs、n 采样、stop、seed 等参数都要在引擎里正确实现。

4.11 总表：技术 → 问题 → 代价

技术	解决	核心思想	代价 / 前提	出处
Continuous batching	P2 P3	按迭代而非按请求调度	需要变长 attention kernel	Orca (OSDI’22)
PagedAttention	P1 P3	KV 分页 + block table	间接寻址的 kernel	vLLM (SOSP’23)

技术	解决	核心思想	代价 / 前提	出处
Prefix caching	P5	KV 跨请求复用	占用显存、需要淘汰策略	SGLang RadixAttention；vLLM APC
KV offload / 分层	P5 P10	KV 下沉 CPU / SSD / 远端	传输带宽、一致性	Mooncake、LMCache、HiCache
Chunked prefill	P4	prefill 切块与 decode 混批	长 prompt 的 TTFT 略增	Sarathi-Serve (OSDI’24)
PD 分离	P4 P13	两阶段放不同 GPU 池	KV 传输、系统复杂度	DistServe、Splitwise、Mooncake
投机解码	P6	猜 k 个、并行验证	大批时变负收益	Leviathan’22；Medusa；EAGLE；MTP
FlashAttention 系列	P9 P10	IO-aware 分块	硬件专用实现	Dao et al. 2022–2025
CUDA graph / 重叠调度	P8	消除 launch 与 CPU 空泡	静态形状、显存占用	vLLM、SGLang、TRT-LLM
量化	P3 P6 P7	少字节	精度风险、需评测	GPTQ、AWQ、SmoothQuant、FP8/FP4
GQA / MLA / 稀疏 / 混合	P3 P10	从结构上减 KV	需要模型训练时决定	GQA’23、DeepSeek-V2’24
TP / PP / EP / CP	P7 P10	切分模型	通信	Megatron、GShard、Ring Attention
Multi-LoRA	P11	共享基座、批内混 adapter	adapter 管理	S-LoRA、Punica
结构化输出	P12	grammar → token mask	mask 计算开销	XGrammar、Outlines、lguidance



第 05 章 · MoE　　专家群岛

05 扩展到 MoE：显存按总参数、计算按激活参数

Mixture-of-Experts 把 FFN 换成 E 个专家，router 为每个 token 选 top-k 个。当前大多数前沿开源模型都是 MoE：

模型	总参数 / 激活参数	专家配置	注意力
DeepSeek-V3 / R1	671B / 37B	256 路由专家 + 1 共享专家，top-8	MLA
Kimi K2	约 1T / 32B	384 专家，top-8	MLA
Qwen3-235B-A22B	235B / 22B	128 专家，top-8	GQA
gpt-oss-120b	约 117B / 5.1B	128 专家，top-4（MXFP4 权重）	GQA + 滑动窗口交错
Mixtral 8×7B	47B / 13B	8 专家，top-2	GQA

MoE：模型特征 → 场景 → 推理负载

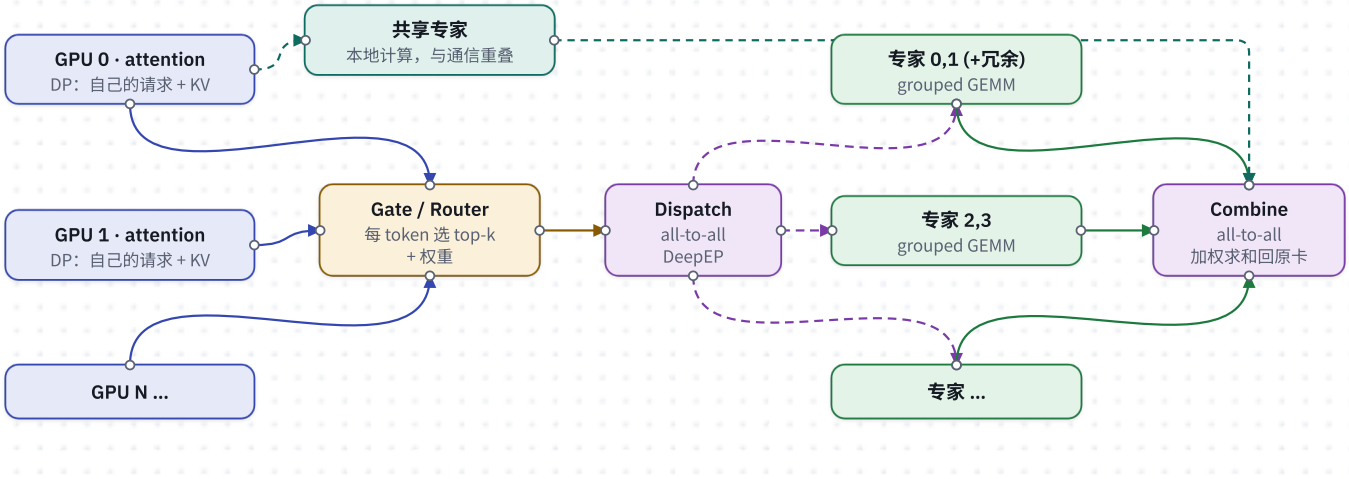
模型特征	为什么这样设计（场景决定）	在推理负载上的体现
稀疏激活：每 token 只用 top-k 个专家	在固定的训练与推理 FLOPs 下扩大参数量（知识容量）	计算按激活参数、显存按总参数；decode 仍要读几乎全部专家权重，但每个专家只分到很少 token → 小而碎的 GEMM
细粒度专家（256、384 个）+ 共享专家	细粒度提高专业化程度，共享专家承载通用知识	每层两次 all-to-all；热门专家导致负载不均 → EPLB 冗余专家；共享专家可本地计算并与通信重叠
MLA（DeepSeek、Kimi）	长上下文 + 高并发需要极小的 KV	KV 小 → 单卡可放更多请求，配合 DP attention 进一步放大 EP 下每专家的批
MTP 模块	训练时提供更密的监督信号	推理时直接当 draft 做投机解码

模型特征	为什么这样设计（场景决定）	在推理负载上的体现
原生 FP8 训练（块级 scale）	降低训练成本	推理直接使用 FP8 权重与块级 scale GEMM（DeepGEMM），无需额外量化

MoE 让推理更难的地方

- **显存：**671B 参数 FP8 也要约 700 GB，至少要 8×H200 或跨机。所有专家都要常驻显存，因为任何 token 都可能路由到任何专家。
- **decode 时专家 GEMM 极小：**批里 B 个 token，每个选 k 个专家，平均每个专家只分到 `B·k/E` 个 token。DeepSeek-V3 在 B=64 时：被激活的专家数 ≈ `E·(1−(1−k/E)^B)` ≈ 222 个（几乎全部），平均每个专家只有约 2.3 个 token。结果是几乎要读全部专家权重，却每个只做极小的矩阵乘——严重 memory-bound。
- **通信：**专家分在不同 GPU 上，每个 MoE 层要两次 all-to-all（dispatch 把 token 发给专家，combine 把结果收回），延迟敏感且量大。
- **负载不均：**热门专家所在的 GPU 成为瓶颈，整个层等最慢的那张卡。训练时可以用 capacity factor 丢 token，推理时不能丢。

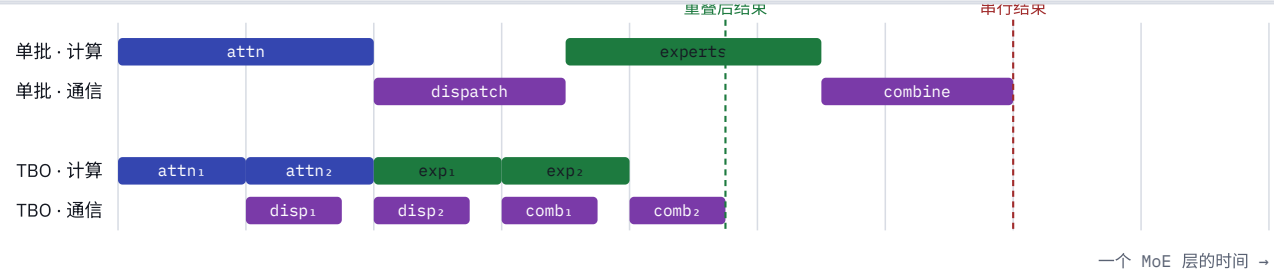
> **关键洞察：用“宽 EP”把每专家批做大**
>
> 如果把 256 个专家分到 128 张卡上，每张卡只放 2 个专家：显存省下来放 KV（并发更多），同时全局所有请求的 token 会汇聚到少数专家上，每个专家的有效批变大，GEMM 回到高效区。这就是 DeepSeek 推理系统的核心思路：**attention 用数据并行（DP，每卡处理自己的请求），MoE 用大规模专家并行（EP）**。



一个 MoE 层的数据流（DP attention + EP experts）。每张卡先对自己的请求做 attention 和 gate；dispatch 用 all-to-all 把 token 发到专家所在的卡；专家做 grouped GEMM；combine 再用 all-to-all 把结果按 gate 权重加回原卡。共享专家在本地算，可与通信重叠。

MoE serving 的技术栈

问题	技术	说明
all-to-all 慢	DeepEP（DeepSeek 开源）	两套 kernel：*normal*（高吞吐，用于 prefill，NVLink + RDMA 转发）和 *low-latency*（纯 RDMA，用于 decode，hook 式通信不占用 SM，便于和计算重叠、兼容 CUDA graph）。同类还有 Perplexity 的 pplx-kernels、基于 NVSHMEM 的实现
通信无法隐藏	Two-Batch Overlap（TBO）/ 微批重叠	把一个批切成两个微批，一个在做 attention / 专家计算时另一个在做 dispatch / combine
专家负载不均	EPLB（Expert Parallel Load Balancer）	根据统计把热门专家复制成冗余副本（DeepSeek prefill 时用 32 个冗余专家），并重新摆放；分层模式（先按节点组均衡）与全局模式；可在线周期性重排
专家 GEMM 小而碎	Grouped GEMM（DeepGEMM、CUTLASS、Triton fused MoE）	DeepGEMM 提供 prefill 用的 contiguous 布局和 decode 用的 masked 布局（形状固定，便于 CUDA graph）；FP8 细粒度 scale
MLA 不好做 TP	DP attention	MLA 的 latent 只有一份，TP 切 head 会复制 KV；改为每卡独立处理不同请求的 attention，KV 不重复
显存不够	FP8 / FP4 专家权重	专家权重占绝大部分，量化收益最大
单卡故障影响整个实例	弹性 EP / 冗余专家	上百卡组成一个推理实例时，故障半径很大，需要专家迁移与快速恢复



Two-Batch Overlap。上：单批时计算和通信串行，GPU 在等网络。下：拆成微批 1、2，一个的通信与另一个的计算重叠，总时间缩短。

源码 `vLLM FusedMoE layer` `vLLM DeepEP low-latency` `DeepEP high-throughput` `vLLM EPLB` `SGLang token_dispatcher` `SGLang two_batch_overlap.py` `SGLang eplb/` `DeepSeek eplb.py` `DeepEP kernels`

案例：DeepSeek-V3/R1 的线上推理系统

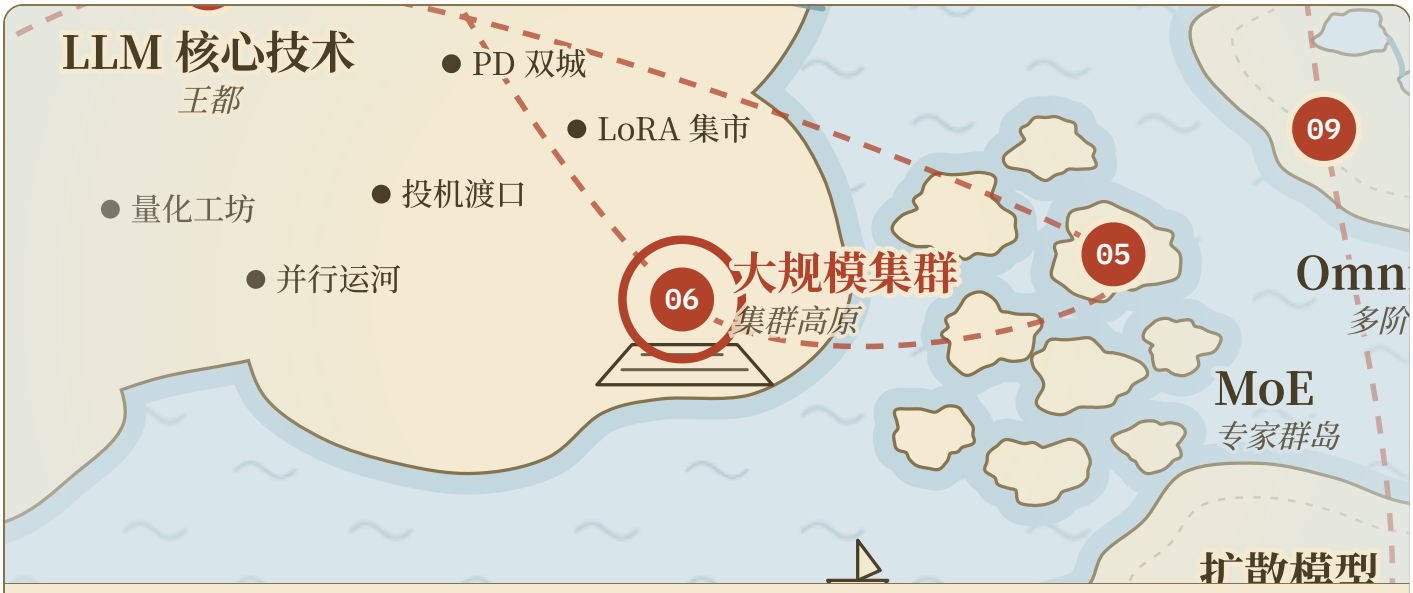
DeepSeek 在 2025 年 2 月的 Open Infra 周最后一天公开了线上系统概况，是面试中讲 MoE serving 最好的例子：

- **PD 分离 + 不同并行度：**prefill 阶段每个部署单元 4 个节点（32 卡），路由专家 EP32、MLA 与共享专家 DP32；decode 阶段每单元 18 个节点（144 卡），EP144、DP144。
- **重叠：**prefill 用双微批重叠，decode 把 attention 切成两段做 5 级流水的重叠。

- **负载均衡：**prefill、decode 和 attention 各有自己的均衡器（专家冗余、按 KV 使用量与请求数分配请求）。
- **结果（公开数字，H800）：**平均每节点 prefill 约 73.7k token/s（含 cache 命中）、decode 约 14.8k token/s；理论成本利润率 545%。
- SGLang 团队随后在 96 张 H100 上用 PD 分离 + 大规模 EP 开源复现，2k 输入时每节点约 52.3k 输入 token/s、22.3k 输出 token/s。

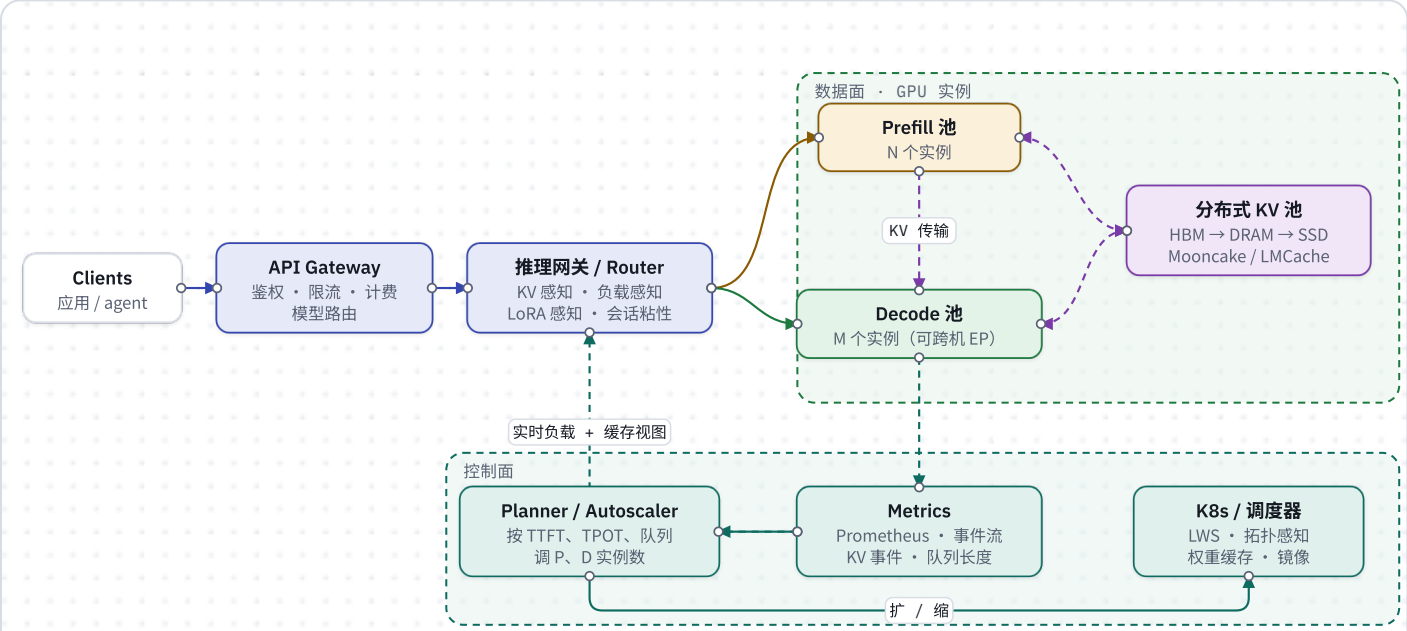
阅读 DeepSeek 推理系统概览 ↗ SGLang 大规模 EP 博客 ↗ DeepEP ↗ DeepGEMM ↗ EPLB ↗

机柜级硬件（GB200 / GB300 NVL72）把 72 卡放进一个 NVLink 域，宽 EP 的 all-to-all 可以完全走 NVLink，是当前超大 MoE 部署的主流形态。



06 扩展到大规模集群

单个引擎实例优化到头之后，问题变成集群层面的：几百上千张 GPU、多个模型、PD 两个池、跨机实例、流量波动、硬件故障。这一层的软件叫 orchestration / control plane，它不替代 vLLM / SGLang，而是管理一群引擎实例。



集群推理的分层。API 网关负责鉴权、限流、计费；推理网关（路由器）按 KV 命中和负载选实例；实例按 PD 分池，跨机实例用 LeaderWorkerSet 等方式编排；分布式 KV 池在实例间共享前缀；控制面根据 SLO 指标扩缩各个池。

路由：命中率和负载均衡的权衡

源码 `Dynamo kv_router` `Dynamo kv-router crate` `llm-d EPP（选实例）` `SGLang Model Gateway（Rust）` `AIBrix gateway` `Gateway API Inference Ext.`

- **轮询 / 最少连接**会把同一会话打散到不同实例，前缀 cache 几乎失效。
- **会话粘性 / 一致性哈希**提高命中，但热点会把某个实例打爆。

- **KV 感知路由**：路由器维护每个实例缓存了哪些 block（通过引擎上报的 KV 事件，或近似 radix tree），对每个候选实例算 “命中可省的 prefill 量 − 负载代价”。代表：NVIDIA Dynamo KV Router、llm-d 的 inference scheduler（prefix-cache scorer）、SGLang Model Gateway（原 sgl-router）的 cache-aware 策略、AIBrix router。
- **Kubernetes 标准化**：Gateway API Inference Extension 定义了 `InferencePool` 和 endpoint picker，把 “模型感知的选实例” 做成 K8s 网关的扩展点，llm-d 等基于它实现。

扩缩容与冷启动

源码 `Dynamo planner` `AIBrix controller` `production-stack` `LeaderWorkerSet`

- **别用 GPU 利用率当信号**：decode 时 nvidia-smi 的 util 几乎总是 100%。应看队列长度、KV 使用率、TTFT / TPOT 的 SLO 违约率。PD 分离时 P 和 D 分开扩（Dynamo Planner 就是做这个）。
- **冷启动链路**：调度 GPU 节点 → 拉镜像（几十 GB）→ 加载权重（几百 GB，从对象存储 → 本地 NVMe → 显存）→ torch.compile / CUDA graph 预热。优化手段：权重预放到节点、并行流式加载（Run:ai Model Streamer、fastsafetensors）、节点间 P2P 分发权重、编译缓存、GPU 进程快照（如 Modal 的 GPU memory snapshot、CUDA checkpoint）。
- **多模型共用 GPU**：vLLM 的 sleep mode（释放权重 / KV 显存，秒级唤醒）、多 LoRA、按需加载，适合长尾模型。

可靠性

- 一个宽 EP 实例跨 18 个节点时，任意一张卡出错整个实例就挂；需要健康检查、快速剔除、请求重试、专家迁移。
- 请求迁移：Llumnix（OSDI’24）在实例间实时迁移请求（连同 KV），用于负载均衡、降级和优雅下线。
- 流量削峰：Batch API（离线任务，24 小时内完成，通常半价）填平低谷；优先级让在线请求先走。

集群层的开源项目

项目	来源	定位	关键能力
NVIDIA Dynamo	NVIDIA（2025 GTC 开源）	引擎无关的分布式推理框架	PD 分离、KV 感知路由、KVBM 多级 KV、NIXL 传输、Planner 自动扩缩；后端可接 vLLM / SGLang / TRT-LLM
llm-d	Red Hat、Google、IBM、NVIDIA、CoreWeave 等	K8s 原生的分布式推理栈	基于 vLLM + Gateway API Inference Extension；前缀感知调度、PD 分离、宽 EP 方案
AIBrix	字节跳动，vLLM 社区	K8s 上的 LLM 控制面	LoRA 管理、LLM 专用 autoscaler、路由、分布式 KV cache、异构 GPU
vLLM production-stack	vLLM 社区 + LMCache	参考部署栈	Helm、路由、LMCache、可观测性
KServe	CNCF	K8s 模型服务标准	InferenceService CRD，LLM 场景集成 llm-d

项目	来源	定位	关键能力
Ray Serve	Anyscale	Python 原生的服务编排	多模型组合、自动扩缩、Ray Serve LLM 封装 vLLM
LeaderWorkerSet	K8s SIG	多节点 Pod 组	一个推理实例跨多机时作为整体调度、整体重启
Mooncake / LMCache	Moonshot + 清华 / LMCache 团队	KV 存储与传输层	Transfer Engine（RDMA）、分布式 KV store；被 vLLM、SGLang、Dynamo 集成
NIXL	NVIDIA	点对点数据传输库	统一 GPU / CPU / 存储之间的异步传输，PD 分离的 KV 传输常用后端



第 07 章 · 推理框架　　框架城邦：vLLM · SGLang

07 LLM 推理框架

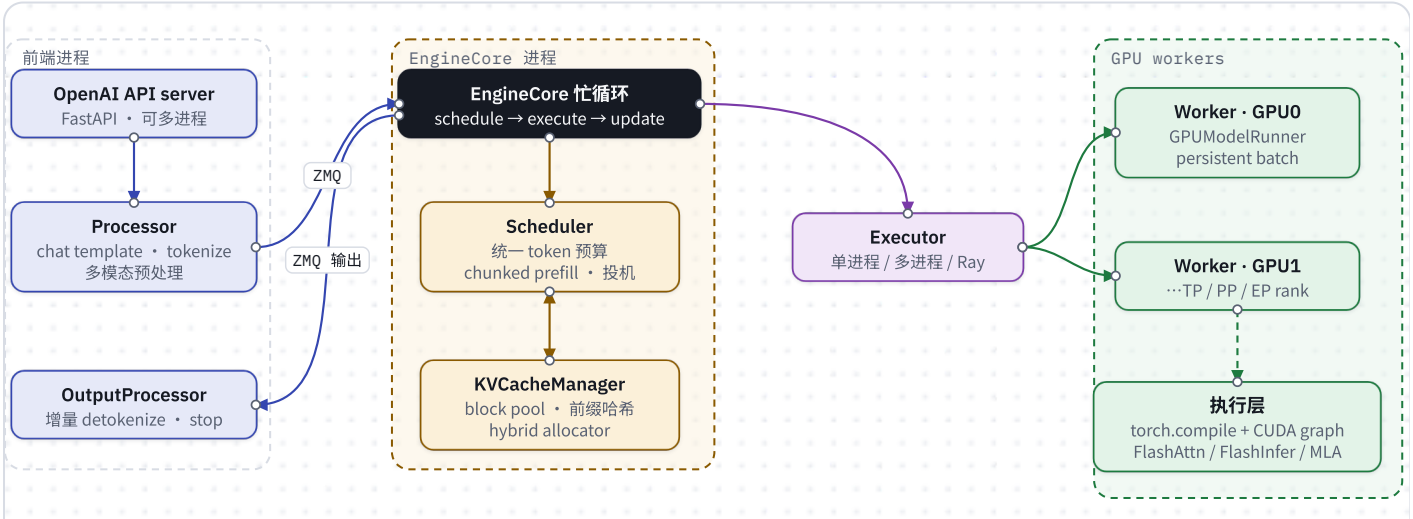
引擎（engine）负责单个实例里的调度、KV 管理和执行；上一节的 orchestration 负责一群实例。
面试最常被问的是 vLLM 和 SGLang 的架构和区别。

vLLM

精读顺序 ① api_server.py ② async_llm.py ③ engine/core.py ④ scheduler.py ⑤ kv_cache_manager.py ⑥ gpu_worker.py ⑦ gpu_model_runner.py ⑧ attention backend

◈ vLLM · Code Wiki

2023 年出自 UC Berkeley Sky Computing Lab（PagedAttention 论文），现为 PyTorch Foundation 托管项目，是模型和硬件覆盖最广的开源引擎。2025 年起 V1 架构成为默认实现。
2026 年 1 月核心作者创立 Inferact（a16z、Lightspeed 领投 1.5 亿美元种子轮）做商业化。



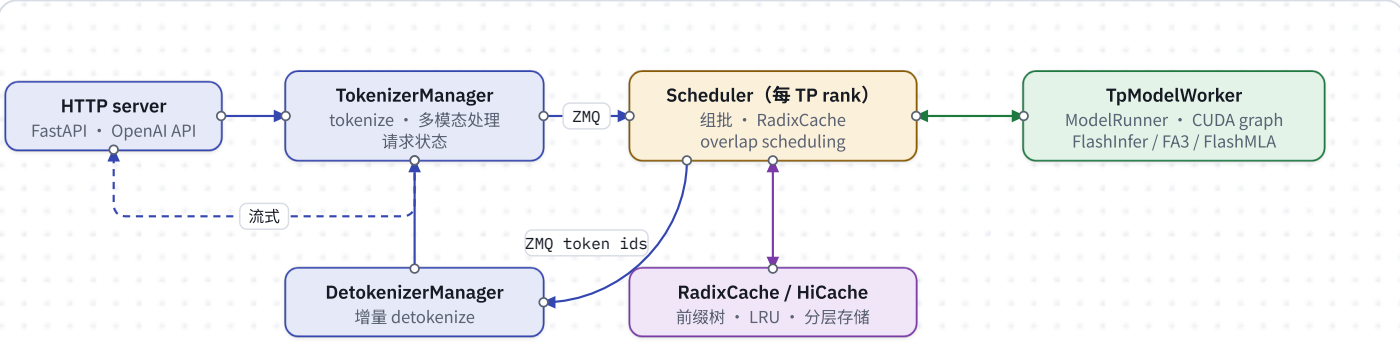
vLLM V1 进程结构。前端进程（可多个 API server）做 HTTP、输入处理（tokenize、多模态预处理）和输出处理（detokenize）；EngineCore 是独立进程，跑一个忙循环：调度 → 执行 → 更新状态；Executor 把每步下发到每张 GPU 上的 Worker。进程间用 ZMQ + msgpack 通信，让 CPU 工作与 GPU 执行并行。

- **特点：**模型覆盖最全（文本、多模态、embedding / reward / pooling）；硬件最广（CUDA、ROCm、TPU、Gaudi、XPU、CPU，以及 vllm-ascend 等插件）；插件化的 KV connector（PD 分离）、attention backend、量化方法；投机解码（EAGLE、MTP、n-gram）；多 LoRA；结构化输出；RL 集成（sleep mode、权重热更新）。
- **周边：**vLLM-Omni（多模态生成）、llm-d、production-stack、AIBrix、llm-compressor、guidellm（压测）、semantic-router（意图路由）。

SGLang

精读顺序 ① http_server.py ② tokenizer_manager.py ③ scheduler.py ④ schedule_policy.py ⑤ radix_cache.py ⑥ tp_worker.py ⑦ model_runner.py ⑧ detokenizer_manager.py ◈ SGLang · Code Wiki

2023 年底出自 LMSYS（Berkeley / Stanford 等），最初是“前端结构化生成语言 + 高效运行时（SRT）”，以 RadixAttention 闻名，后来在 DeepSeek 大规模 EP、结构化输出、RL rollout 方面成为主力引擎之一。2026 年核心团队成立 RadixArk（Accel 领投），继续维护开源 SGLang 并开发 RL 框架 Miles。



SGLang 运行时进程结构。TokenizerManager 在主进程里接 HTTP 请求并 tokenize，通过 ZMQ 发给 Scheduler；每个 GPU（TP rank）一个 Scheduler 进程，内部管理 RadixCache、组批、调用 ModelRunner；输出 token 发给 DetokenizerManager，再回到 TokenizerManager 流式返回。重叠调度让 CPU 准备下一批与 GPU 计算并行。

- **特点：**RadixAttention 与 cache-aware 调度；零开销 overlap scheduler；DP attention + DeepEP / DeepGEMM / EPLB / TBO 的大规模 EP；PD 分离（Mooncake / NIXL）；EAGLE-2/3 投机；HiCache 分层 KV；sgl-kernel 自有 kernel 库；SGLang Model Gateway（Rust 写的路由器）。
- **RL 生态：**verl、slime、AReaL、Miles 等都把 SGLang 作为 rollout 引擎（见 08）。
- **扩展：**SGLang Diffusion（图像 / 视频生成）、SGLang-Omni（多阶段语音与 omni 模型）、TPU 后端（SGLang-JAX）。

vLLM vs SGLang：面试怎么答

维度	VLLM	SGLANG
起点	PagedAttention，解决 KV 显存	RadixAttention + 前端 DSL，解决复杂 LLM 程序的前缀复用
前缀缓存	block 哈希（APC），默认开启	radix tree，cache-aware 调度
调度	V1 统一 token 预算；async scheduling	overlap scheduler（CPU / GPU 流水）

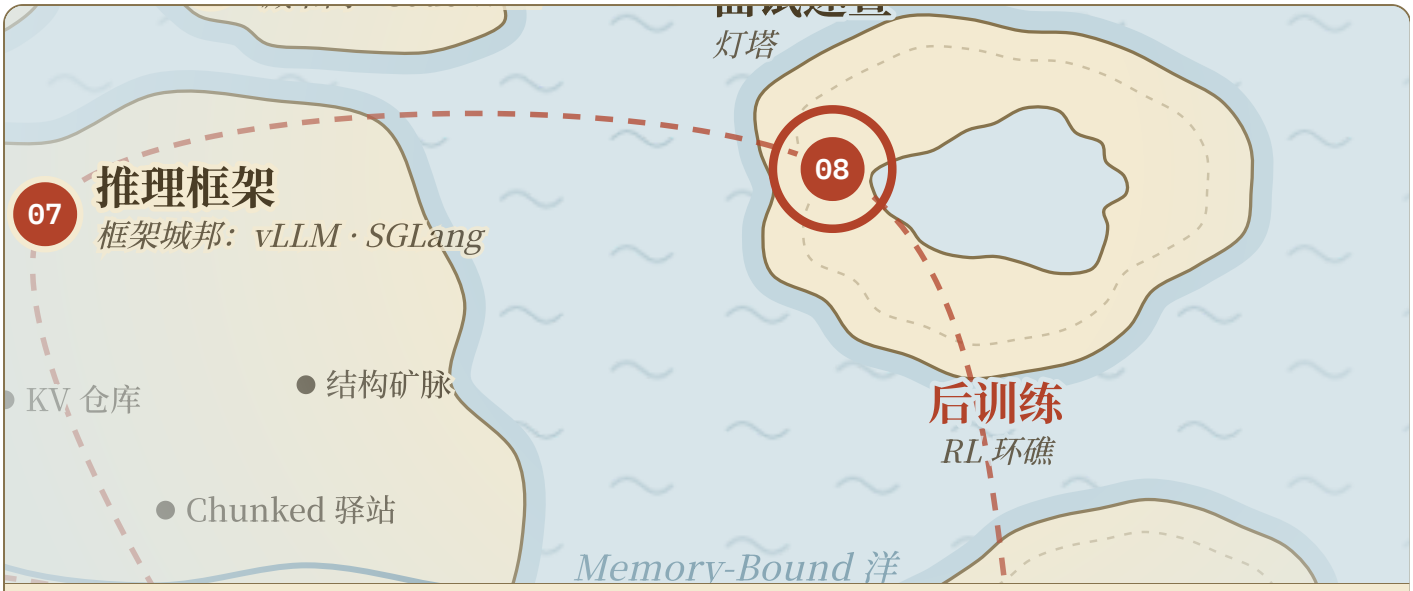
维度	VLLM	SGLANG
强项	模型与硬件覆盖、插件生态、企业部署（Red Hat、llm-d）	DeepSeek 类 MoE 大规模部署、结构化输出、RL rollout
共同点	continuous batching、paged KV、chunked prefill、PD 分离、投机解码、CUDA graph、FP8/FP4、多 LoRA、OpenAI API；都大量使用 FlashInfer、FlashAttention、DeepGEMM 等共享 kernel。两者的性能差距随版本和负载变化，结论应来自自己负载上的基准测试。	

其他引擎

源码 `TRT-LLM py_executor.py` `resource_manager.py` `llama.cpp kv-cache` `llama.cpp server`

`◈ TRT-LLM · Code Wiki` `◈ llama.cpp · Code Wiki`

引擎	来源	定位与特点
TensorRT-LLM	NVIDIA	NVIDIA 硬件上 kernel 最激进（XQA attention、Blackwell FP4、wide EP）；早期需要编译 TensorRT engine，1.0 起以 PyTorch 后端为主；与 Dynamo、Triton Inference Server 集成。只支持 NVIDIA
LMDeploy	上海 AI Lab	TurboMind（C++）+ PyTorch 双引擎；W4A16 量化与 VLM 支持好
TGI	Hugging Face	开源 continuous batching 的早期实现之一（Rust 路由 + Python 模型）；据报道 2025 年底进入维护模式，官方引导新部署使用 vLLM / SGLang
llama.cpp / Ollama	社区	本地与边缘：CPU、Apple Silicon、Vulkan；GGUF 量化；Ollama 在其上做模型管理与易用 API。并发能力有限
MLX	Apple	Apple Silicon 上的数组框架，mlx-lm 用于本地推理
MLC-LLM	CMU / OctoAI 背景	基于 TVM 编译到多后端，包括浏览器 WebGPU（WebLLM）和手机
KTransformers	清华 MADSys	CPU + GPU 异构：专家放 CPU 内存、attention 放 GPU，单机跑超大 MoE
DeepSpeed-FastGen	Microsoft	Dynamic SplitFuse（与 chunked prefill 思路相同），目前较少更新
Modular MAX	Modular	Mojo 写的跨硬件推理引擎
TPU 系	Google 等	JetStream / MaxText、vLLM TPU 后端、SGLang-JAX
国内厂商自研	阿里、华为、百度等	RTP-LLM（阿里）、MindIE（华为昇腾）、FastDeploy（百度飞桨）等，通常针对自家硬件和模型
Triton Inference Server	NVIDIA	通用模型服务器（多框架、ensemble），LLM 场景下作为 TRT-LLM 的前端；现归入 Dynamo 品牌（Dynamo-Triton）

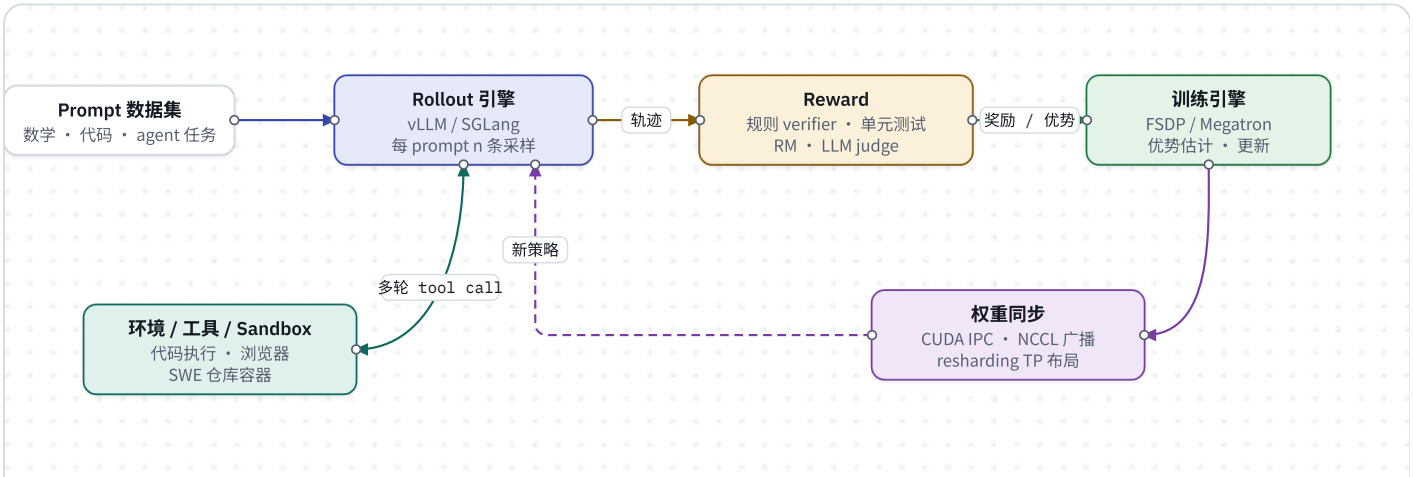


第 08 章 · 后训练　　RL 环礁

08 后训练里的推理：RL rollout 是一个推理问题

推理引擎不只用来服务用户，后训练的很多环节本质上都是大批量推理：

- **在线 RL**（PPO、GRPO、DAPO 等）：每一步都要用当前策略为每个 prompt 生成多条回答（GRPO 常见 8–64 条），推理模型的回答动辄上万 token。
- **数据合成与蒸馏**：用强模型批量生成 SFT 数据；rejection sampling / best-of-N 挑选高分样本。
- **打分**：reward model、LLM-as-judge 也是推理负载。
- **训练中的评测**：每隔若干步在基准上跑一遍。



RL 训练闭环。rollout 引擎（vLLM / SGLang）生成回答，agentic 任务还要和 sandbox 中的工具 / 环境多轮交互；verifier 或 reward model 打分；训练引擎（FSDP / Megatron）更新参数；新权重同步回 rollout 引擎，进入下一步。紫色虚线是权重同步，它每一步都要发生。

后训练：模型特征 → 场景 → 推理负载

模型特征	为什么这样设计（场景决定）	在推理负载上的体现
推理模型输出长 CoT（几千到几万 token）	用可验证奖励训练推理能力，思考越充分越可能答对	rollout 极度 decode-heavy；长度长尾 → step 时间由最长样本决定

模型特征	为什么这样设计（场景决定）	在推理负载上的体现
GRPO 类组采样：每 prompt 采 n 条	用组内相对奖励代替 critic 估计优势	n 条共享整段 prompt → 前缀 cache 命中极高；批很大，吞吐优先
策略每一步都更新	on-policy 学习	权重高频同步；推理引擎要能热更新、释放 / 收回显存
多轮工具使用（agentic）	训练 agent 在真实环境里完成任务	环境步骤与 sandbox 成为关键路径；多轮 KV 复用；异步调度
MoE 策略 + FP8 rollout	前沿模型规模与成本	训练与推理的路由、数值不一致 → off-policy 修正、路由回放、确定性 kernel

和在线服务相比，rollout 有什么不同

维度	在线服务	RL ROLLOUT
目标	满足每个请求的 SLO	一个训练 step 的总时间最短（看最慢的那条）
权重	长期不变	每一步都更新，需要快速同步
数值	输出合理即可	logprob 要和训练引擎一致，否则变成 off-policy
长度分布	大多较短	长尾极重：少数回答长达几万 token
前缀共享	系统 prompt	同一 prompt 的 n 条采样共享整段 prompt（前缀 cache 收益大）
显存	独占 GPU	常与训练共用 GPU（colocate），要能快速让出 / 收回显存

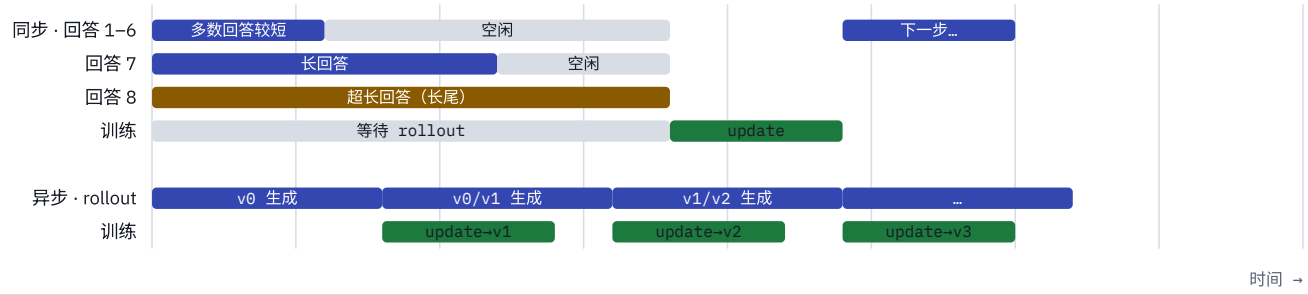
四个核心难题

1. 权重同步

- **Colocated（共置）：**训练和推理分时共用同一批 GPU。训练时推理引擎“睡眠”（vLLM sleep mode / SGLang `release_memory_occupation` 释放 KV 和权重显存），推理时把训练状态 offload。权重通过 CUDA IPC 句柄零拷贝传给推理引擎，并从 FSDP / Megatron 的切分方式 reshard 成推理的 TP 布局。verl 的 HybridEngine 是代表。
- **Disaggregated（分离）：**训练和推理各用一组 GPU，权重通过 NCCL / RDMA 广播。资源利用更灵活，适合异步 RL；超大模型要优化传输（例如 Moonshot 开源的 checkpoint-engine 用于万亿参数模型的快速权重更新）。

2. 长尾（straggler）

同步 RL 里，一个 step 必须等最长的回答生成完，其他 GPU 空转。



同步 vs 异步。上：同步 RL，rollout 引擎大部分时间在等两条超长回答，训练也只能等。下：异步 RL，rollout 持续生成并把完成的样本流式交给训练，训练用“落后一两个版本”的数据，引擎在 step 之间更新权重。

- **Partial rollout**（Kimi k1.5）：超长回答先截断，下一轮接着生成。
- **异步 / off-policy**：AReaL（全异步 + staleness 控制）、verl / slime 的 one-step off-policy、PipelineRL（生成中途更新权重）。
- **动态采样 / 过采样**：DAPO 丢掉奖励全相同（无梯度信号）的组，多发一些 prompt 来凑满批。

3. 训练–推理不一致（training–inference mismatch）

推理引擎算出的 token 概率和训练引擎对同一序列重算的概率不完全一样：kernel 不同、批大小不同导致的归约顺序不同、rollout 用 FP8 而训练用 BF16、MoE 的路由在两边选了不同专家。结果是“名义上的 on-policy 实际上是 off-policy”，会导致训练不稳定甚至崩溃。

- **训练侧重算 logprob**，并用重要性采样修正（如截断重要性采样 TIS）。
- **MoE 路由回放**：记录 rollout 时每个 token 用了哪些专家，训练时强制用相同路由。
- **确定性 / 批不变（batch-invariant）kernel**：Thinking Machines 2025 年的文章《Defeating Nondeterminism in LLM Inference》指出非确定性主要来自 kernel 的归约顺序随批大小变化，而不是常说的“GPU 并发”；之后 vLLM 和 SGLang 都加入了确定性推理模式（会损失一些性能）。

4. Agentic RL 的多轮交互

- 一条轨迹包含多次“生成 → 调工具 → 拿结果 → 再生成”，环境步骤（跑单元测试、编译、浏览网页）可能比生成还慢，需要异步调度，否则 GPU 空等。
- 需要大量并发 sandbox（几千个隔离的容器 / microVM），见 14。
- 多轮之间要复用 KV（会话粘性、前缀 cache）；尽量 “token in, token out”，避免每轮把文本重新 tokenize 带来的 token 序列漂移。

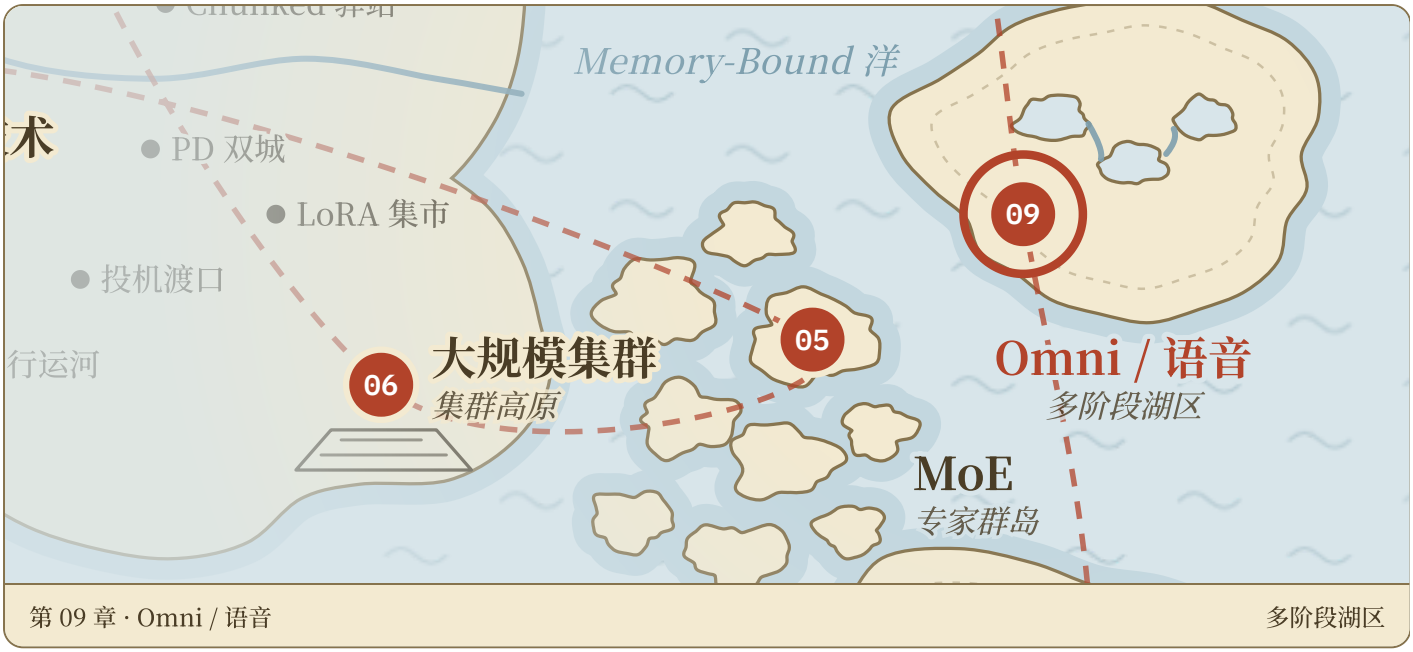
RL 框架与它们用的推理引擎

源码 `verl rollout/` `verl checkpoint_engine` `slime backends/` `AReaL areal/` `TRL vllm_client.py` OpenRLHF `SGLang weight_sync/` `vLLM sleep mode` ◈ verl · Code Wiki

框架	来源	训练后端	ROLLOUT 引擎	特点
verl	字节 Seed（HybridFlow 论文）	FSDP / Megatron	vLLM / SGLang	单控制器 + 多控制器混合编程模型；共置 HybridEngine；生态最大。另有基于 vLLM-Omni 的 VeRL-Omni，支持多模态 / 扩散模型的 RL

框架	来源	训练后端	ROLLOUT 引擎	特点
OpenRLHF	社区	DeepSpeed	vLLM	基于 Ray，分离式部署
TRL	Hugging Face	Transformers / Accelerate	vLLM（colocate 或 server 模式）	易用，GRPOTrainer 等
slime	清华 / 智谱	Megatron	SGLang	GLM-4.5 系列的 RL 框架；同步 / 异步模式
Miles	RadixArk	Megatron 等	SGLang	面向大规模 MoE 的企业级 RL 框架
AReaL	蚂蚁 / 清华	FSDP / Megatron	SGLang / vLLM	全异步 RL，生成与训练完全解耦
ROLL	阿里	Megatron / FSDP	vLLM / SGLang	大规模、agentic 场景
NeMo-RL	NVIDIA	Megatron / DTensor	vLLM	NVIDIA 生态
SkyRL / prime-rl	Berkeley NovaSky / Prime Intellect	FSDP 等	vLLM / SGLang	长程 agent 任务 / 去中心化异步训练

商业形态：Thinking Machines 的 Tinker 把“训练 + 采样”做成 API；各家推理平台也提供微调 + 托管一体的服务。

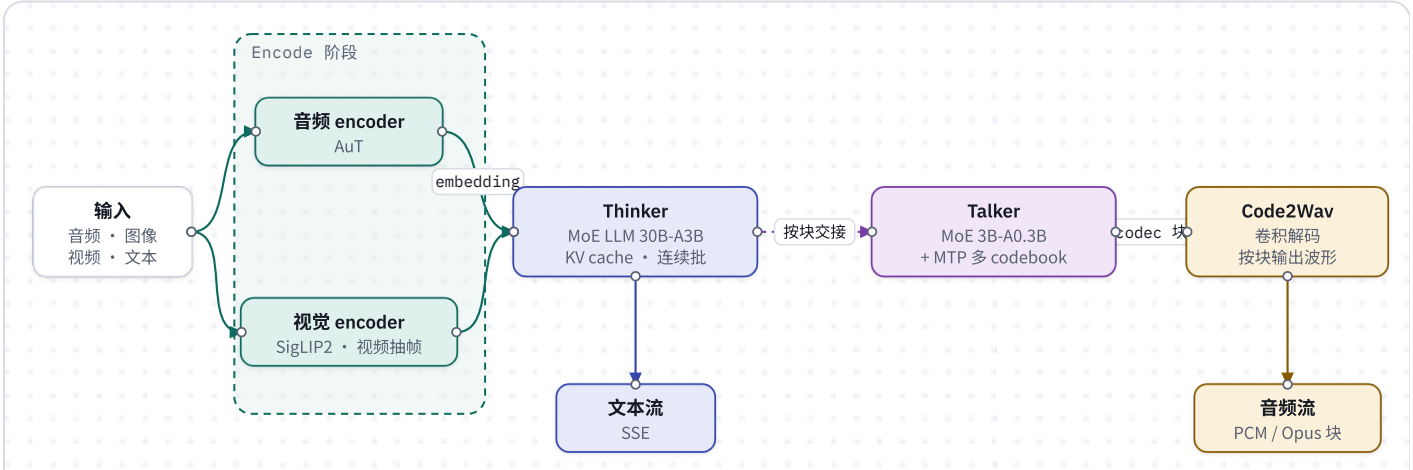


09 Omni / 语音 / 多模态模型的推理服务

多模态分两类：**多模态输入**（VLM，图像 / 视频 / 音频 → 文本），基本可以沿用 LLM serving，只多一个 encoder；**多模态输出**（文本 + 语音、图像生成、any-to-any），模型由多个异构组件串起来，serving 的假设被根本改变。

代表模型的结构：Thinker–Talker

以 Qwen3-Omni 为例：音频 encoder（AuT）和视觉 encoder（SigLIP2）把输入编码后交给 **Thinker**（30B 总参数 / 3B 激活的 MoE LLM）生成文本；Thinker 的隐藏状态再交给 **Talker**（3B / 0.3B 激活的 MoE，带 MTP 预测多层 codebook）生成音频 codec token；最后 **Code2Wav**（约 2 亿参数的卷积网络）把 codec token 解码成波形，按块流式输出。类似结构还有 Qwen2.5-Omni、MiniCPM-o、Ming-Omni 等；全双工模型如 Kyutai 的 Moshi 能边听边说。



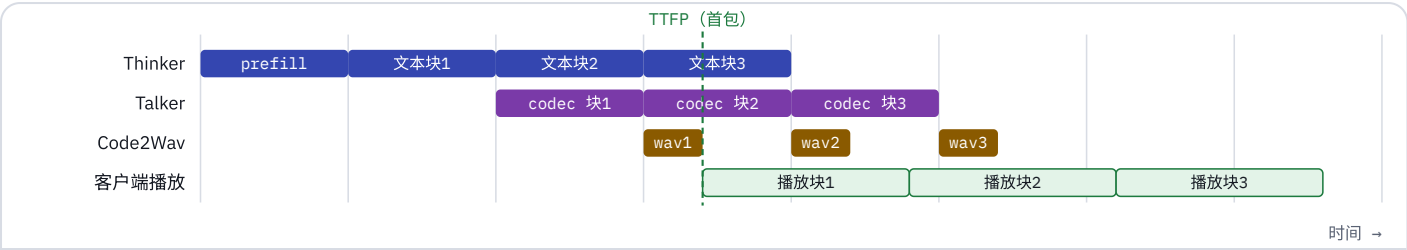
多阶段 omni 流水线（以 Qwen3-Omni 为例）。每个阶段的计算特征完全不同：encoder 是非自回归的大算子，Thinker 是标准 LLM decode，Talker 是小模型高频 decode，Code2Wav 是卷积解码。阶段之间按块（chunk）异步交接，让第一段音频尽早出来。

Omni / 语音：模型特征 → 场景 → 推理负载

模型特征	为什么这样设计（场景决定）	在推理负载上的体现
统一的多模态输入（音频 / 视觉 encoder）	一个模型理解所有模态	encoder 是非自回归的大算子；一张图几百到几千个视觉 token → prefill 变长；encoder 输出可缓存，或拆成独立 encode 池（EPD）
Thinker–Talker 分离	文本推理和语音生成需要不同规模的模型，解耦后各自最优	一个请求跨多个异构阶段，每个阶段的批大小与资源不同；阶段间要传隐藏状态
离散音频 codec：多 codebook（RVQ），每秒十几到几十帧	用语言模型的方式生成音频	每秒音频需要几十次 Talker decode × 多个 codebook → 小模型、超高频 decode，launch 开销占比高，需要 CUDA graph / MTP 一次预测多层
因果、可流式的 codec 解码器	实时对话	chunk 级流水；首包延迟（TTFP）和 RTF < 1 成为硬约束
全双工（边听边说）	自然的对话体验、可打断	持久会话 + 固定节拍推进 + 打断时取消；调度从“吞吐优先”变成“实时优先”

相对 LLM serving，哪些假设变了

变化	带来的问题	做法
一个模型 = 多个异构阶段	不同阶段的最优批大小、并行方式、显存需求不同；一个阶段慢会拖住整条流水线	**阶段化运行时**：每个阶段有自己的调度器和执行器（AR 阶段复用 vLLM / SGLang 引擎，非 AR 阶段用轻量调度），阶段可以共置或放到不同 GPU，按负载分配资源
阶段间要传张量	隐藏状态、codec token、embedding 在进程 / GPU / 机器之间流动	传输层抽象：共享内存、NCCL、NIXL、Mooncake（vLLM-Omni 的 OmniConnector；SGLang-Omni 的 relay 数据面）
输出是实时流	用户关心第一段音频多快出来，以及之后会不会断（播放 underrun）	阶段间 **chunk 级异步交接**：Talker 不等 Thinker 说完；Code2Wav 每凑够一块 codec token 就解码（带左上下文重叠避免接缝）
多模态预处理重	图像 / 视频解码和缩放在 CPU 上；一张图几百到几千个视觉 token	预处理多进程化、encoder 输出缓存（同一张图多轮复用）、前缀 cache 把多模态内容哈希进 key、**EPD 分离**（encoder 单独成池）
全双工实时交互	模型边听边说；用户随时打断；每个会话要按固定节拍（如每 80 ms）推进	持久会话（会话状态常驻 GPU）、WebSocket / WebRTC 实时 API、VAD 与轮次管理、打断时取消生成并回收 KV；调度更像实时系统
TTS 特有结构	多 codebook（RVQ）、参考音频声音克隆	delay pattern / depth transformer、小模型 decode 用 CUDA graph、说话人 embedding 缓存



流式重叠。Thinker 还在生成文本时，Talker 已经开始处理第一块；Code2Wav 解码出第一块音频后客户端立即播放。首包延迟（TTFP）由“第一块”路径决定，而不是整句；之后只要每块的生成速度快于播放速度（RTF < 1），播放就不会断。

级联 vs 端到端

级联：ASR → LLM → TTS

模块成熟、可替换、易调试；延迟是三段相加（每段都要流式才能压下来）；语气、情绪等副语言信息在 ASR 处丢失。框架：Pipecat、LiveKit Agents 负责 VAD、轮次、打断和 WebRTC 传输。

端到端 omni / 全双工

延迟更低、能保留副语言信息、能边听边说；但服务端复杂（多阶段、持久会话、实时节拍），评测也更难。代表：GPT-4o / Realtime API、Gemini Live、Qwen-Omni、MiniCPM-o、Moshi。

指标

TTFT（文本首 token）、**TTFP / TTFA**（首个音频包）、TPOT、**RTF**（生成耗时 ÷ 音频时长，< 1 才能实时）、播放 underrun；质量上用 ASR 转写生成语音算 WER（可懂度）、说话人相似度（SIM）、UTMOS（自然度）；ASR 服务看 WER 和 RTFx（每秒处理多少秒音频）。

框架

项目	定位
vLLM-Omni	vLLM 官方项目（2025-11 发布），阶段化流水线，AR + DiT 混合，OmniConnector 全解耦，覆盖 omni、TTS、扩散、机器人策略模型，支持全双工实时
SGLang-Omni	SGLang 生态的多阶段语音 / omni 运行时：阶段专属调度、relay 数据面（shm / NCCL / NIXL / Mooncake）、全双工与级联语音助手、ASR / TTS / 说话人分离
vLLM / SGLang / LMDeploy 原生 VLM	图像 / 视频 / 音频输入 → 文本，encoder 缓存、多模态前缀 cache
NVIDIA Riva / NeMo、faster-whisper、whisper.cpp	ASR / TTS 专用推理
Pipecat、LiveKit Agents	语音 agent 编排层：VAD、轮次、打断、WebRTC，后端接各种 ASR / LLM / TTS

源码 `vLLM-Omni engine/` `OmniConnector` `core/` `架构文档` `SGLang-Omni pipeline/` `stage_io.py` `kv_transfer.py` `models/` ◈ vLLM-Omni · Code Wiki ◈ SGLang-Omni · Code Wiki

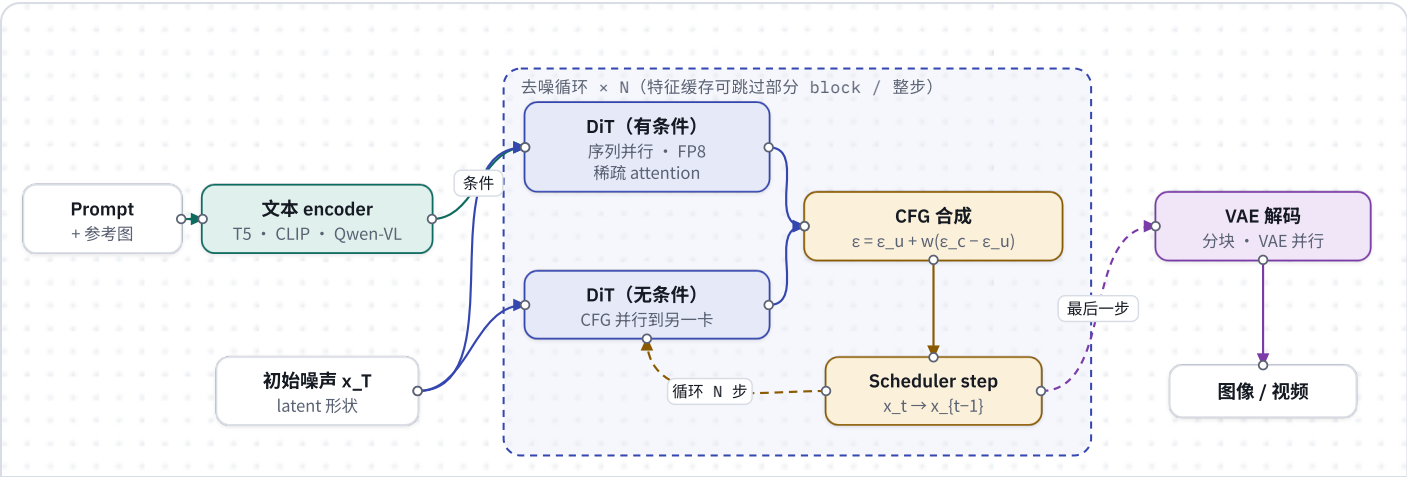


第 10 章 · 扩散模型　　去噪湾

10 扩散模型推理

图像 / 视频 / 音频扩散模型（DiT 为主，如 FLUX、SD3、Qwen-Image、Wan、HunyuanVideo）的推理和 LLM 几乎是镜像关系：

维度	LLM DECODE	扩散
迭代方式	每步生成 1 个 token，步数 = 输出长度（未知）	每步对**全部**潜变量去噪，步数固定（常见 20–50，蒸馏后 1–8）
瓶颈	显存带宽	**算力**（大 GEMM + 超长序列的 attention）
状态	KV cache 随长度增长	无 KV（双向注意力）；步与步之间特征相似，可做特征缓存
序列长度	上下文 token	视频极长：Wan 720p、81 帧经 VAE（时间 4×、空间 8×）和 2×2 patch 后为 21×45×80 = 75,600 个 token
延迟	毫秒级 / token	图像数秒，视频数十秒到数分钟
批处理收益	巨大	有限（单请求已经 compute-bound），主要用于小图
额外开销	—	CFG 让每步算两遍（有条件 + 无条件）；视频 VAE 解码显存极大



扩散推理流水线。文本 encoder 只跑一次；去噪循环跑 N 步，每步（使用 CFG 时）算有条件和无条件两次 DiT；特征缓存会在相邻步变化小时跳过部分 block；最后 VAE 把潜变量解码成像素。每个环节旁边标了对应的加速手段。

扩散：模型特征 → 场景 → 推理负载

模型特征	为什么这样设计（场景决定）	在推理负载上的体现
双向注意力的 DiT，作用于整个潜变量	图像 / 视频需要全局一致的结构，所有位置互相可见	没有 KV cache，每一步都是对全部 token 的“prefill” → compute-bound；单请求就能吃满 GPU，批处理收益有限
迭代去噪 N 步（扩散 / flow matching）	从噪声逐步细化，质量与多样性好	延迟 = N × 单步；相邻步特征相似 → 特征缓存；蒸馏把 N 降到个位数
Classifier-free guidance	提高与 prompt 的对齐程度	每步算两遍（有 / 无条件）→ CFG 并行，或 guidance 蒸馏去掉无条件分支
高压缩比时空 VAE（8×8 空间、4× 时间）	把像素降到可训练的 token 数	视频仍有数万 token → 序列并行（Ulysses / Ring）与稀疏注意力；VAE 解码本身是显存峰值
大文本编码器（T5-XXL、VLM）	理解长而复杂的 prompt	额外数 GB 到数十 GB 显存；只跑一次，可单独部署或 offload

扩散推理的加速技术

类别	方法	说明
减少步数	更好的采样器；蒸馏：LCM / 一致性模型、ADD（SDXL-Turbo）、DMD / DMD2、FLUX.1-schnell 的少步模型	收益最大（50 步 → 4 步），但需要训练，可能损失多样性
特征缓存（免训练）	DeepCache、FORA、Δ-DiT、TeaCache、First-Block Cache、TaylorSeer；库：Cache-DiT	相邻步的中间特征很相似，按阈值复用上一步的 block 输出或残差；常见 1.5–2× 加速、轻微质量损失
序列并行	DeepSpeed-Ulysses（all-to-all 换 head）、Ring Attention、两者组合的 USP（xDiT）	视频生成的主力并行方式，attention 占主导时近线性加速

类别	方法	说明
其他并行	CFG 并行（有 / 无条件放两张卡）、PipeFusion（patch 级流水，利用相邻步激活的相似性）、DistriFusion（patch 并行 + 异步过期激活）、TP、VAE 并行	按模型规模和分辨率组合
注意力	FlashAttention-3、SageAttention（INT8 / FP8 量化 attention）；视频稀疏注意力：Sliding Tile Attention、Sparse VideoGen、Radial Attention	长视频序列下 attention 占 70%+ 时间，稀疏化收益大
量化	FP8 DiT、SVDQuant / Nunchaku（W4A4 + 低秩分支吸收离群值）、NVFP4	同时省显存和算力
编译	torch.compile、TensorRT、CUDA graph；融合 adaLN 调制、norm	小分辨率时 CPU 开销显著
显存	按层 CPU offload、VAE 分块 / 分片解码	让 14B–20B 模型跑在消费级显卡上

扩散的 serving 系统设计

- **异步任务 API**：生成要数十秒，通常是“提交 → 轮询 / webhook / 流式进度”，而非长连接等待。
- **请求级 vs 步级批处理**：每一步跑的是同一个网络，不同请求处在不同步也能拼批（step-level continuous batching）；要求分辨率 / 帧数相同或做分桶。vLLM-Omni 等已实现扩散的连续批处理。
- **LoRA 热切换**：图像平台上用户大量使用风格 LoRA，需要快速加载与融合。
- **流水线解耦**：文本 encoder、DiT、VAE 可以放不同 GPU / 不同批大小（AR + DiT 混合模型更需要，如 HunyuanImage-3、BAGEL）。
- **与 LLM 技术融合**：自回归视频扩散（CausVid、Self-Forcing）按帧块生成并缓存 KV，能实时流式出视频；交互式世界模型（如 Genie 3、Cosmos 类）把扩散推理变成了“流式 + KV cache + 低延迟”的问题。扩散语言模型（LLaDA、Mercury、Gemini Diffusion）则把并行去噪带到了文本生成。

指标：单图 / 单视频端到端延迟、每秒步数（it/s）、吞吐（images/s）、峰值显存；流式视频看首帧时间；质量看 FID、CLIP score、GenEval、VBench 与人类偏好分。

框架

项目	定位
Diffusers	参考实现与模型库，几乎所有扩散模型首发支持
ComfyUI	节点式工作流，创作者生态核心，常被包装成 API 服务
xDiT	DiT 并行推理引擎：USP、PipeFusion、CFG 并行
FastVideo	视频生成加速：Sliding Tile Attention、蒸馏、序列并行
LightX2V	视频生成推理框架：量化、缓存、步数蒸馏组合

项目	定位
Nunchaku / **Cache-DiT**	4-bit SVDQuant 推理 / 统一的 DiT 特征缓存库
SGLang Diffusion、vLLM-Omni	把 LLM serving 的工程能力（OpenAI 兼容 API、批处理、并行、量化）带到图像 / 视频生成
TensorRT / ModelOpt	NVIDIA 的编译与量化路径

源码 `Diffusers pipelines/` `DiT 实现` `hooks（缓存 / offload）` `xDiT core/` `FastVideo attention/` `cache-dit caching/` `Nunchaku csrc` `SGLang Diffusion runtime` `vLLM-Omni diffusion/`

◈ Diffusers · Code Wiki ◈ xDiT · Code Wiki



11 机器人推理：从“吞吐”回到“实时”

机器人策略模型（VLA，Vision-Language-Action）在结构上是“VLM + 动作头”，看起来像多模态 LLM，但推理的目标函数完全不同：它运行在一个物理闭环里，**延迟和抖动直接决定任务成败与安全**，而吞吐几乎无所谓。

代表模型

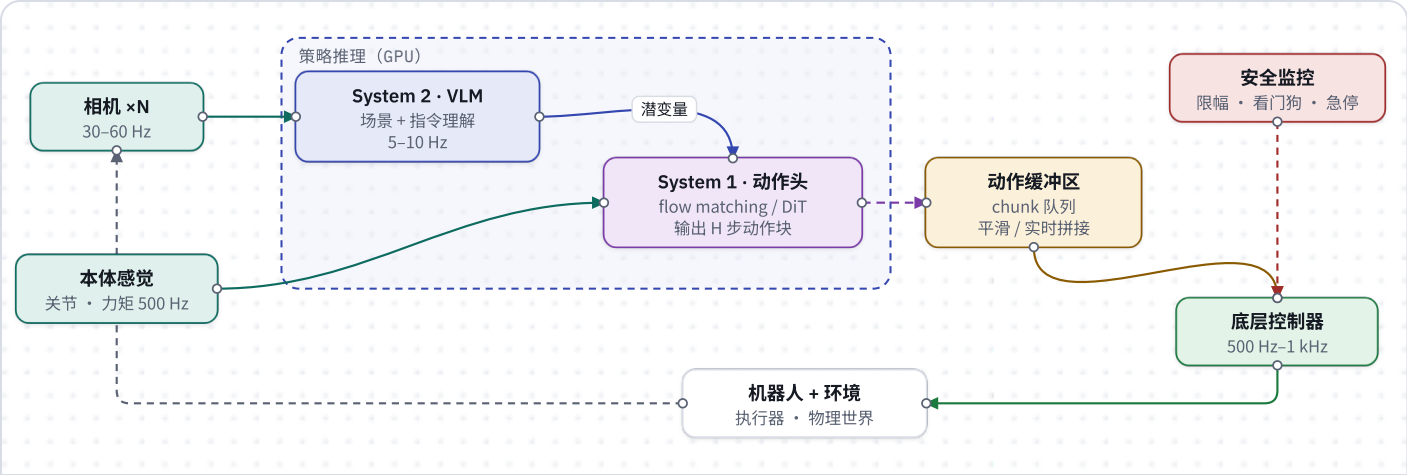
模型	结构	推理相关特点
RT-2（Google DeepMind，2023）	大 VLM 直接输出离散化动作 token	模型很大，控制频率只有个位数 Hz，依赖云端
OpenVLA（2024）	7B，Llama-2 骨干，自回归输出动作 token	每个动作维度一个 token，逐个生成，频率低
π0 / π0.5（Physical Intelligence）	PaliGemma 3B VLM + 约 3 亿参数的 action expert，flow matching	一次推理输出 50 步的动作块（action chunk），用约 10 步积分生成连续动作，支持 50 Hz 控制
GR00T N1 系列（NVIDIA）	System 2：VLM 理解（约 10 Hz）；System 1：DiT 动作头（flow matching）	快慢双系统；面向人形机器人，部署到 Jetson Thor
Helix（Figure）	S2：7B VLM，7–9 Hz；S1：约 8000 万参数视觉运动策略，200 Hz	双系统解耦，S1 用 S2 的最新潜变量
Gemini Robotics	云端大模型骨干 + 机器人本地的动作解码器	云端负责理解，本地负责高频输出
SmolVLA（Hugging Face）	约 4.5 亿参数的小 VLA	消费级硬件可跑，配合异步推理

机器人：模型特征 → 场景 → 推理负载

模型特征	为什么这样设计（场景决定）	在推理负载上的体现
VLM 骨干 + 动作专家（action expert）	用互联网规模的视觉语言知识泛化，同时输出精细的连续控制	每次推理 = 一次图像 + 指令的 prefill + 动作头多步积分；图像每帧都变，前缀几乎无法复用，prefill 主导延迟
动作块（一次输出 H 步，如 50 步）	动作更平滑，降低对推理频率的要求	推理频率可以低于控制频率，但必须异步执行并在块之间平滑衔接（RTC）
flow matching / 扩散动作头	动作分布是多峰的（同一场景有多种合理动作）	积分步数直接计入延迟（π0 约 10 步）；步数与平滑度之间要权衡
多相机 + 本体感觉输入	空间理解与精确定位	每次推理多张图 → 视觉 token 多；传感器数据要低延迟进入 GPU
快慢双系统（S2 / S1）	语义理解可以慢，运动控制必须快	两个模型按不同频率运行、可放在不同硬件上，需要多速率调度

机器人推理 vs LLM serving

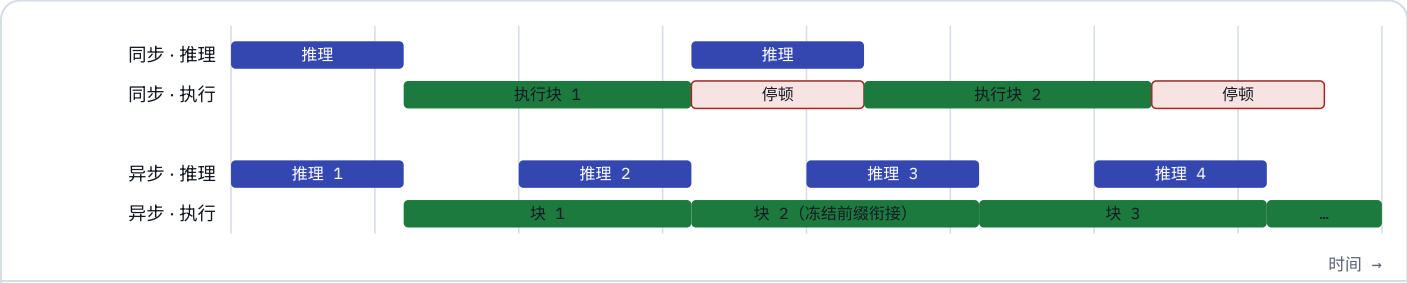
维度	LLM SERVING	机器人推理
闭环	开环：生成完交给用户	闭环：动作改变世界，下一帧观测依赖上一步动作；推理慢 = 用过期的观测做决策
批大小	越大越好	每台机器人 batch = 1（机队云端推理时可以跨机器人拼批）
关心的统计量	p50 / p99 延迟、吞吐	**最坏情况延迟、抖动**、是否错过控制周期
硬件	数据中心 GPU	机载边缘设备（Jetson Orin / Thor，功耗几十到上百瓦）或经 Wi-Fi / 5G 访问远端 GPU（网络抖动）
频率	按请求	多速率：VLM 推理 5–10 Hz，动作策略 50–200 Hz，底层关节控制 500 Hz–1 kHz
失败代价	重试即可	不能重试，可能造成碰撞；需要看门狗、动作限幅、超时回退
输入	文本 / 图片	多路相机流 + 本体感觉（关节角、力矩），持续不断



多速率控制回路。慢系统（VLM）理解场景和指令，快系统（动作头）输出动作块，放进动作缓冲区；底层控制器以高频从缓冲区取动作驱动电机；安全监控可以随时截断。外环和内环的频率差一到两个数量级。

关键技术

- **Action chunking**（ACT，2023）：一次推理预测未来 H 步动作，降低推理频率要求；相邻块重叠部分用 temporal ensembling 平滑。π0 系列用 50 步的块。
- **异步推理**：执行当前块的同时就开始推理下一块，机器人不需要停下来等模型（LeRobot 的 PolicyServer + RobotClient 架构，SmolVLA 论文中提出）。
- **Real-Time Chunking（RTC，Physical Intelligence 2025）**：异步时新块和正在执行的块会“接不上”。RTC 把生成新块看作 inpainting：推理延迟期间必定会执行的动作被冻结，其余部分被引导与之衔接，在不重新训练的前提下得到平滑连续的动作。
- **快慢双系统**：把慢的语义理解和快的运动控制解耦，各自按自己的频率运行（Helix、GR00T）。
- **压缩与编译**：INT8 / FP8 / INT4 量化、TensorRT 引擎、更少的 flow 积分步、蒸馏小策略；batch = 1 时 kernel launch 开销占比高，CUDA graph / JAX jit 很关键。
- **远程推理**：机器人只做传感与控制，策略跑在附近的 GPU 上（openpi 用 websocket 提供策略服务）；要处理网络抖动和断连回退。
- **仿真与世界模型**：训练和评测依赖大规模 GPU 仿真（Isaac Sim / Isaac Lab）和视频世界模型（NVIDIA Cosmos 等），它们本身也是推理负载（大批量、扩散生成）。



同步 vs 异步执行。同步：每次推理时机器人停下等待，动作不连贯。异步（+ RTC）：推理与执行重叠，延迟期间执行旧块中已承诺的动作，新块从衔接点接上，运动连续。

框架

项目	定位
LeRobot	Hugging Face 的机器人学习库：数据集格式、ACT / Diffusion Policy / π0 / SmolVLA 等策略、异步推理服务
openpi	π0 / π0.5 的开源实现（JAX / PyTorch），自带 websocket 策略服务器
Isaac GR00T	NVIDIA 人形机器人基础模型，含 TensorRT 部署到 Jetson 的流程
Isaac ROS / Holoscan、ROS 2	机载传感器流水线与机器人中间件（DDS 通信、实时调度）
dora-rs	Rust 写的低延迟机器人数据流框架，零拷贝共享内存
vLLM-Omni	已把 π0.5、GR00T 等机器人策略模型纳入统一的服务运行时（云端 / 机队场景）

源码 `LeRobot policy server` `openpi websocket_policy_server.py` `GR00T deployment/` `dora-rs`

`◈ openpi · Code Wiki` `◈ LeRobot · Code Wiki`

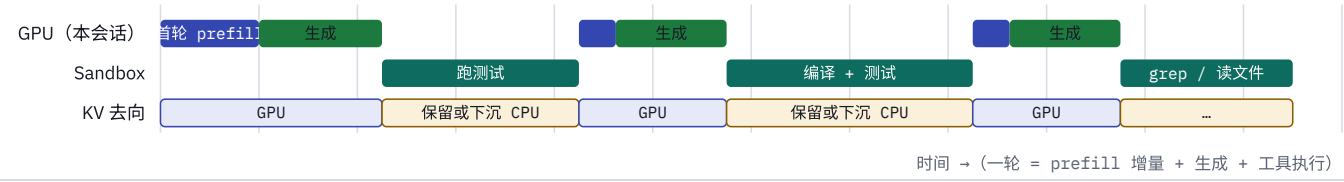


12 Coding Agent 的推理：长会话、高命中、慢工具

代码是目前 LLM 商业化最成功的场景之一，但“写代码”背后其实是三种完全不同的推理负载：编辑器内的**行内补全**、把建议应用到文件的**快速编辑（apply）**，以及自主完成任务的 **agentic coding**（Claude Code、Codex、Cursor Agent、OpenHands 等）。它们对 serving 的要求差别之大，几乎可以当成三个产品来设计。

模型特征 → 场景 → 推理负载

模型特征	为什么这样设计（场景决定）	在推理负载上的体现
超长上下文（128k–1M token）	要读懂整个仓库、长日志、长 diff；agent 任务一次跑几十上百轮	单请求 prompt 常在 50k–200k；KV 显存巨大（70B 级模型 100k 上下文单请求 30 GB 量级）；prefill 是主要算力消耗
工具调用 + 在 SWE 类环境上做 agentic RL	模型要自己读文件、改代码、跑测试、看报错再修	**多轮、追加式上下文**：每轮只在末尾追加工具输出，前面全部不变 → 前缀命中率可达 90% 以上，成为最关键的指标；两轮之间有工具执行的“空档”（秒到分钟）
推理（thinking）模式，交错思考与工具调用	规划、定位 bug、复杂重构需要先想再做	输出也长；agent 循环是串行的，每用户 tokens/s 直接决定任务完成时间，所以 TPOT 比聊天更重要
FIM（fill-in-the-middle）训练的小模型	编辑器补全需要看到光标前后的代码	补全：输出很短、端到端预算几百毫秒、QPS 极高、大量请求在完成前就被取消（用户继续打字）
代码高度可预测，编辑大量复制原文	改代码时大部分内容保持不变	投机解码收益特别大：prompt lookup / n-gram 直接从原文件取 draft（“speculative edits”），apply 模型可以跑到每秒上千 token
大 MoE 骨干（Qwen3-Coder、Kimi K2、GLM、DeepSeek、MiniMax 系列等）	强能力 + 可控推理成本	继承 05 的所有 MoE 部署问题；长上下文让 KV 而不是权重成为显存主体



一次 agentic coding 会话的推理时间线。每一轮都是“很长的前缀（几乎全命中）+ 新的工具输出（需要 prefill）+ 一段生成”，轮与轮之间 GPU 在等工具执行。如果这段空档里 KV 被淘汰，下一轮就要重算整个前缀；所以 KV 要么留在 GPU、要么下沉到 CPU / SSD 并能快速取回，路由还必须把同一会话送回同一实例（或能取到同一份 KV）。

三种负载的对比

维度	行内补全（TAB）	快速编辑 / APPLY	AGENTIC CODING
模型	小模型（几 B），FIM	中小模型，专门训练“改写文件”	前沿大模型（常为 MoE），thinking + tool use
输入 / 输出	几千 token / 几十 token	整个文件 + 修改意图 / 几乎整个文件	5 万–20 万 token（逐轮增长）/ 几百到几千 token 每轮
延迟目标	端到端几百毫秒以内	整文件改写一两秒	每轮 TTFT 与 TPOT；整体任务完成时间（分钟级）
关键技术	小模型 + 低延迟部署、请求取消、边缘 / 近端部署、前缀 cache（同文件反复请求）	投机解码（原文件作 draft）、高 tokens/s	前缀 cache 与 KV 分层存储、会话粘性 / KV 感知路由、长上下文 kernel、tool-call 解析、sandbox 编排、上下文压缩（compaction）
计费与成本	按席位	按 token	缓存命中的输入远便宜于未命中；命中率直接决定毛利

Serving 侧的具体做法

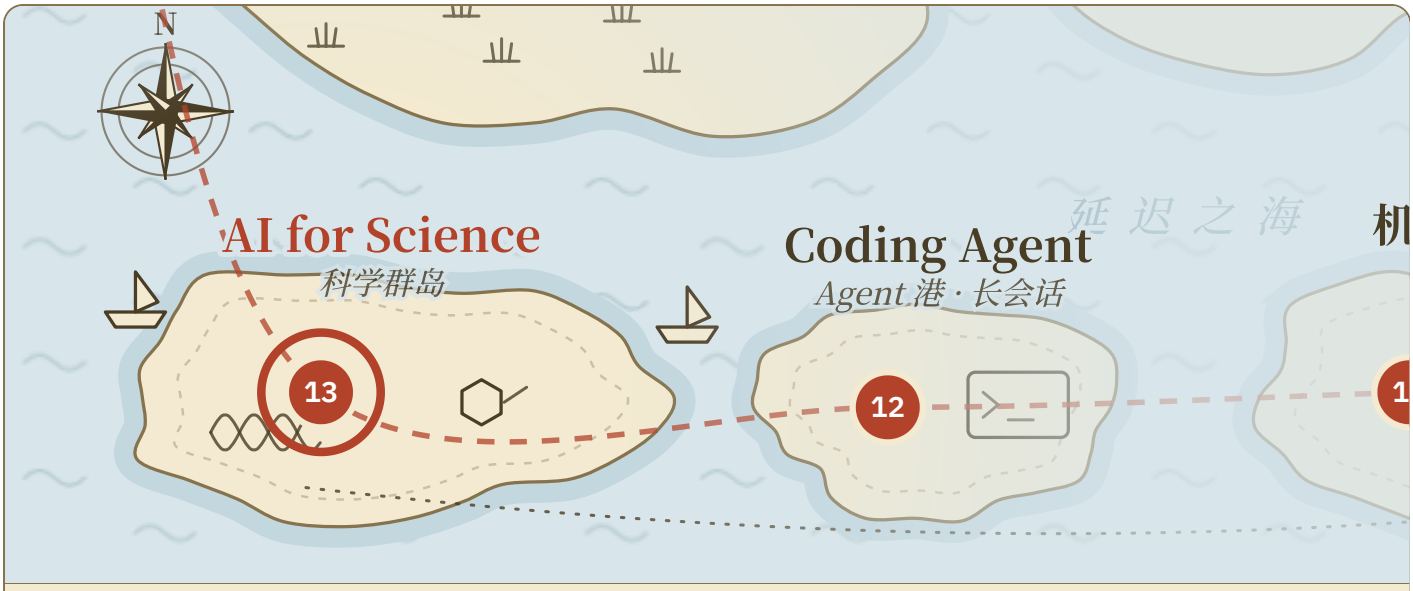
- **把命中率当一等指标。**监控 per-session 的 cached-token 比例；路由用会话 ID 或前缀哈希做粘性，再叠加负载感知（06 的 KV 感知路由）。商业 API 的 prompt caching（例如 Anthropic 默认 5 分钟、可选 1 小时 TTL 的缓存）就是把“工具空档里保留 KV”这件事产品化。
- **KV 的生命周期跨越工具调用。**GPU 上用 LRU 保留，压力大时下沉到 CPU / SSD（LMCache、HiCache、Mooncake、Dynamo KVBM）；按 4.2 的估算，载回远比重算便宜。
- **长上下文 prefill 的成本控制：**chunked prefill 防止一个 200k prompt 卡住别人；必要时用上下文并行；agent 侧做上下文压缩、子 agent 拆分任务，减少单会话上下文长度。
- **可靠的工具调用：**每个模型家族的 tool-call 格式不同，引擎里的 tool parser / reasoning parser 必须和模型模板严格匹配；必要时用结构化输出约束参数 JSON。
- **Sandbox 是推理链路的一部分：**每个会话一个隔离环境（容器 / microVM），要快速创建、可暂停恢复、可快照；训练（agentic RL）和线上用的是同一套。阿里开源的 OpenSandbox 就是面向 coding agent、GUI agent、评测和 RL 的通用 sandbox 平台（Docker / Kubernetes 运行时，多语言 SDK，内置 MCP server），见 14。
- **评测：**SWE-bench 系列（真实仓库 issue → 补丁 → 在容器里跑测试）、Terminal-Bench 等；评测本身就是“推理 + 大量 sandbox”的负载。

源码 `vLLM tool_parsers/` `vLLM reasoning/` `SGLang function_call/` `SGLang reasoning_parser.py`

n-gram draft（编辑场景） LMCache

Agent / Sandbox OpenSandbox（阿里开源） OpenSandbox execd OpenSandbox server SDKs

K8s 运行时 SWE-ReX deployment SWE-bench harness OpenHands ◈ OpenSandbox · Code Wiki



第 13 章 · AI for Science　　科学群岛

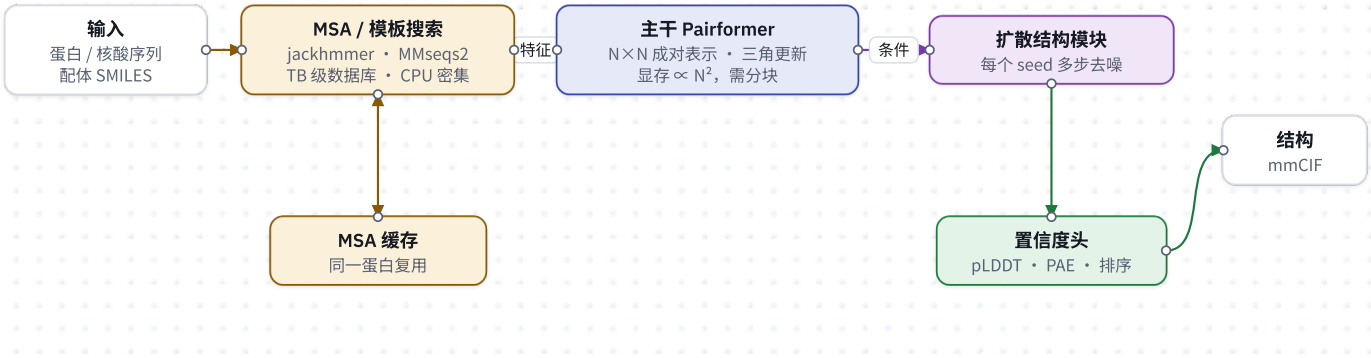
13 AI for Science 的推理：结构、几何与物理

科学模型的输入不是自然语言，而是蛋白质序列与多序列比对、原子坐标、基因组、全球气象网格。这些数据的结构决定了模型结构（成对表示、等变网络、图网络、超长卷积），进而让推理负载和 LLM serving 很不一样：**batch 往往为 1、显存随问题规模平方增长、CPU 侧预处理可能比 GPU 推理还慢、对数值精度更敏感、大量是离线批量筛选。**

模型特征 → 场景 → 推理负载

模型特征	为什么这样设计（科学问题决定）	在推理负载上的体现
蛋白 / 复合物结构预测（AlphaFold2/3、OpenFold、Boltz、Chai-1、Protenix）：维护 N×N×c 的成对表示，用三角乘法 / 三角注意力更新	结构本质是残基两两之间的几何关系，三角更新近似强制“三角不等式”一致性	显存随残基数 **平方**增长，三角注意力计算接近 **立方**；几千残基的复合物就会爆显存 → 分块（chunking）、低精度、专用三角注意力 kernel；每个复合物单独推理，batch≈1，形状各不相同（难以 CUDA graph）
以 MSA（多序列比对）和模板为输入	同源序列的共进化信号揭示哪些残基在空间上接触	推理前要在 TB 级序列库上做 jackhmmer / HHblits 搜索，CPU 小时级，常常比 GPU 推理慢得多 → MMseqs2（ColabFold）加速、MSA 缓存与共享服务、GPU 加速搜索
AF3 类模型的扩散结构模块 + 多 seed 采样与置信度排序	全原子（蛋白、核酸、小分子、离子）生成，需要多样本评估置信度	一次预测 = 主干一次 + 扩散几百步 × 多个样本 → 吞吐与延迟都看扩散部分；可以像 10 一样批量化多个样本
蛋白 / DNA 语言模型（ESM-2/ESM3、Evo 2）：Transformer 或卷积–注意力混合（Evo 2 的 StripedHyena 2），上下文最长到百万碱基	基因组有长程调控关系；进化数据量巨大，适合自监督	最像 LLM 的科学负载：主要是**批量 embedding / 打分**（整个蛋白组、变异效应评分）而不是交互生成 → 吞吐优先、离线；长序列需要长卷积 / 上下文并行 kernel

模型特征	为什么这样设计（科学问题决定）	在推理负载上的体现
天气 / 气候模型（GraphCast：网格上的 GNN；FourCastNet：频域算子；GenCast：扩散集合预报）	全球格点、物理场之间的相互作用；预报本身是概率性的	自回归滚动几十步（例如 6 小时一步滚 10 天）；集合预报要几十个成员 → 天然的批；输入输出是 TB 级再分析数据，**I/O 与数据管线**常是瓶颈；需要在 HPC 定时批量运行
机器学习原子势（MLIP）（MACE、NequIP / Allegro、Meta 的 UMA）：E(3) 等变图网络，球谐与张量积	物理量在旋转 / 平移下必须正确变换（能量不变，力等变），保证分子动力学稳定	嵌在分子动力学循环里跑**百万步**，每步一个小图（几百到几千原子）→ 延迟与 kernel launch 主导；等变张量积 kernel 是热点（cuEquivariance）；常要求 FP32 甚至 FP64 精度，BF16 可能破坏能量守恒
科研 agent（文献、实验规划、调用仿真工具）	把 LLM 当作科研助理	本质上和 12 的 coding agent 一样：长上下文、多轮工具调用、sandbox 执行代码



AlphaFold3 类结构预测的推理流水线。CPU 侧的序列库搜索常常占端到端时间的大头；GPU 侧主干（Pairformer）的显存随 N^2 增长，扩散模块要为每个 seed 跑多步去噪。服务化时通常把 MSA 搜索、主干推理、扩散采样拆成独立的、可分别扩缩的阶段。

与 LLM serving 的差异总结

维度	LLM SERVING	AI FOR SCIENCE 推理
批处理	continuous batching，几百并发	常为 batch=1（结构预测、MD）；或天然大批（集合预报、蛋白组 embedding、虚拟筛选）
形状	变长但每 token 计算相同	问题规模差别巨大（几十到几千残基），显存按 N^2 增长，常需按大小分桶 / 分配不同显存的卡
瓶颈	decode 带宽	成对表示的显存、三角运算、CPU 预处理（MSA）、数据 I/O、小图上的 launch 开销
精度	FP8 / FP4 已普遍	BF16 / TF32 为主，MLIP 等可能需要 FP32 / FP64；可重复性要求高
部署形态	在线 API	HPC 批作业（Slurm）、科研平台的异步任务、少量在线服务（如 AlphaFold Server、NVIDIA BioNeMo NIM）
专用 kernel	attention、GEMM、MoE	三角注意力 / 三角乘法、带 pair bias 的 attention、等变张量积、球谐、图的 scatter / gather、FFT

- **优化手段：**主干分块与 activation offload；三角注意力 kernel（NVIDIA cuEquivariance 提供 triangle attention / multiplication，DeepSpeed4Science 有 Evoformer attention）；MSA 搜索 GPU 化与缓存；扩散样本批量化；MLIP 用 CUDA graph 和融合的等变 kernel 降低每步延迟，并与 LAMMPS / ASE 等 MD 引擎原生集成。
- **服务化：**NVIDIA BioNeMo（框架与 NIM 微服务，覆盖结构预测、蛋白 / 基因组语言模型）、Google DeepMind 的 AlphaFold Server、ColabFold 的公共 MSA 服务、NVIDIA Earth2Studio（天气模型推理与集合预报流水线）、ECMWF 的 AI 预报业务化运行。

结构预测 AF3 evoformer.py | AF3 diffusion_head.py | AF3 数据流水线（MSA） | OpenFold evoformer.py | Boltz pairformer.py | Boltz 三角注意力 | Chai-1 | Protenix model/ | ColabFold batch.py

序列模型 ESM-2 modules.py | ESM3 | Evo 2 models.py | BioNeMo recipes models/

天气 / 原子 GraphCast（weathernext） | Earth2Studio 预报模型 | MACE blocks.py | MACE × LAMMPS | UMA（fairchem） | cuEquivariance triangle | 等变张量积 | e3nn

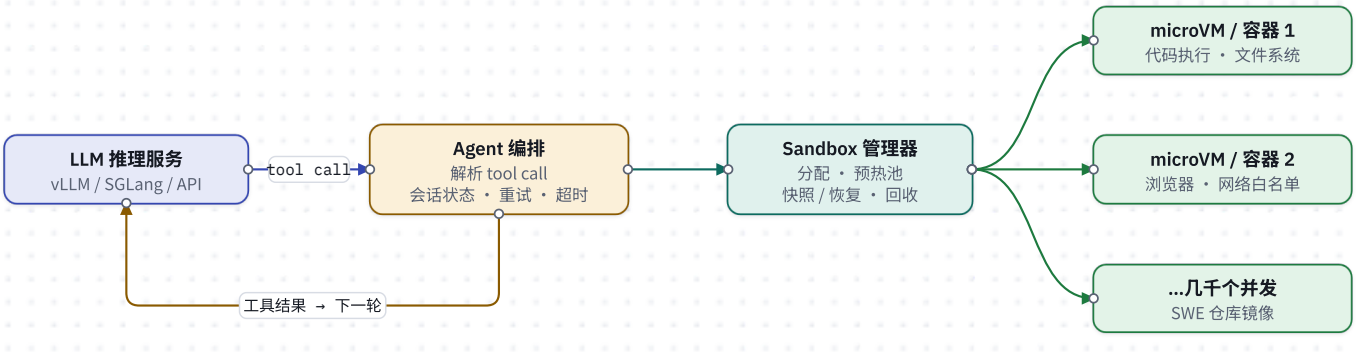
Code Wiki ◈ AlphaFold3 | ◈ Boltz | ◈ MACE | ◈ Earth2Studio



14 和 inference serving 相关的技术

Sandbox：让模型生成的代码和动作安全地执行

Agent 会写代码、跑命令、开浏览器；RL 训练需要几千个并发环境跑单元测试。这些代码不可信，必须在隔离环境里执行。sandbox 已经成为 agent 产品和 RL 基础设施的标准组件。



Agent 循环中的 sandbox。模型输出工具调用，编排层把它路由到 sandbox 池中属于该会话的实例执行，结果作为下一轮输入。预热池和快照恢复负责把“开一个新环境”的时间压到毫秒到百毫秒级。

隔离技术	原理	启动 / 开销	隔离强度	谁在用
容器（Docker）	Linux namespace + cgroups，共享宿主内核	最快、最省	最弱（内核漏洞可逃逸）	受信任代码、内部 RL 环境
gVisor	用户态内核拦截系统调用	较快，系统调用密集型负载有开销	中高	Modal、Google Cloud Run

隔离技术	原理	启动 / 开销	隔离强度	谁在用
Firecracker microVM	基于 KVM 的极简 VMM（AWS Lambda 的底座）	约百毫秒级启动，单实例内存开销仅数 MB，支持快照恢复	高（硬件虚拟化）	E2B 等 agent sandbox 平台
OpenSandbox（阿里开源）	每个 sandbox 是一个 OCI 容器，注入 Go 写的执行守护进程 execd 负责命令、文件、代码执行；统一的 sandbox API + 多语言 SDK + MCP server	Docker 本地 / Kubernetes 集群运行时，可接入更强隔离的容器运行时	取决于底层运行时	coding agent、GUI agent、agent 评测、RL 训练环境
Kata Containers	每个容器一个轻量 VM	中	高	K8s 多租户
WebAssembly / V8 isolate	语言运行时级隔离	毫秒级	中（能力受限）	浏览器内 Python（Pyodide）、边缘函数

- **关键需求：**冷启动（预热池、快照 / 恢复、CRIU）、密度（单机几百上千个）、网络出口控制、文件系统持久化与暂停 / 恢复（长时 agent 会话）、资源限额与超时、可观测性。
- **开源平台：**OpenSandbox（阿里开源，仓库现位于 opensandbox-group 组织）提供统一 sandbox API、Python / Java / JS / C# / Go SDK、CLI 与 MCP server，并带有 Claude Code、Gemini CLI、Codex CLI 等 coding agent 的接入示例；E2B 也开源了 SDK 与基础设施代码。
- **服务商：**E2B、Modal Sandboxes、Daytona、Runloop、Northflank、Cloudflare Sandbox / Containers、Vercel Sandbox 等；模型厂商自带的代码执行工具（OpenAI Code Interpreter、Anthropic code execution tool）。
- **开源：**Firecracker、gVisor、Kata；面向 agent 的运行时如 OpenSandbox、OpenHands runtime、SWE-ReX。

源码 `OpenSandbox` `execd 守护进程` `server` `kubernetes` `examples（各 coding CLI 接入）` `E2B packages/` `Firecracker 快照 API` `gVisor runsc/sandbox` `gVisor 系统调用实现` `SWE-ReX` `◈ OpenSandbox · Code Wiki` `◈ Firecracker · Code Wiki`

API 网关、模型路由与缓存

- **统一网关：**LiteLLM（开源代理，统一上百家供应商的 OpenAI 兼容接口）、OpenRouter（模型市场与路由）、Portkey、Kong AI Gateway、Envoy AI Gateway、Cloudflare AI Gateway。负责鉴权、按 token 限流（TPM / RPM）、配额、计费、重试与 fallback。
- **模型路由：**按问题难度选模型以省钱（RouteLLM、vLLM Semantic Router）；按地域、价格、延迟选供应商。
- **缓存：**精确缓存、语义缓存（向量相似度命中，需谨慎）、供应商侧的 prompt caching。
- **多租户安全：**前缀缓存可能被用作跨租户的时序侧信道（命中更快），可以按租户加 cache salt 隔离（vLLM 支持 `cache_salt`）。

源码 `LiteLLM router.py` `LiteLLM proxy/` `vLLM Semantic Router`

可观测性、压测与评测

类别	工具	用途
引擎指标	vLLM / SGLang 的 Prometheus 指标、NVIDIA DCGM	队列长度、KV 使用率、TTFT / ITL 直方图、前缀命中率、GPU 功耗与温度
链路追踪	OpenTelemetry（GenAI 语义约定）	每个请求的排队、prefill、decode 分段耗时
LLM 应用观测	Langfuse、LangSmith、Helicone、Arize Phoenix	prompt、成本、质量追踪
性能剖析	PyTorch Profiler、Nsight Systems / Compute	找 GPU 空泡、慢 kernel、CPU 瓶颈
压测	`vllm bench serve`、`sglang.bench_serving`、NVIDIA AIPerf（GenAI-Perf 后继）、guidellm、LLMPerf	按到达率 / 并发扫描，画吞吐–延迟曲线
行业基准	MLPerf Inference（MLCommons）、SemiAnalysis InferenceMAX、Artificial Analysis	硬件 / 软件栈对比、API 供应商速度与价格对比
精度评测	lm-evaluation-harness、OpenCompass、EvalScope	量化、投机、新 kernel 上线前的回归

源码 `guidellm` `AIPerf` `lm-evaluation-harness` `vLLM benchmarks/` `sglang bench_serving.py`

模型加载、分发与冷启动

- **格式：**safetensors（可 mmap、零拷贝、无反序列化风险）、GGUF（llama.cpp）。
- **加速加载：**Run:ai Model Streamer、fastsafetensors、CoreWeave tensorizer、GPUDirect Storage；权重预置到节点本地 NVMe；节点间 P2P 分发（如 Dragonfly）。
- **镜像：**推理镜像动辄几十 GB，用懒加载镜像（eStargz、Nydus）或预拉取。
- **快照：**进程级 / GPU 显存级快照恢复，把“加载 + 编译 + 预热”的分钟级冷启动压到秒级。

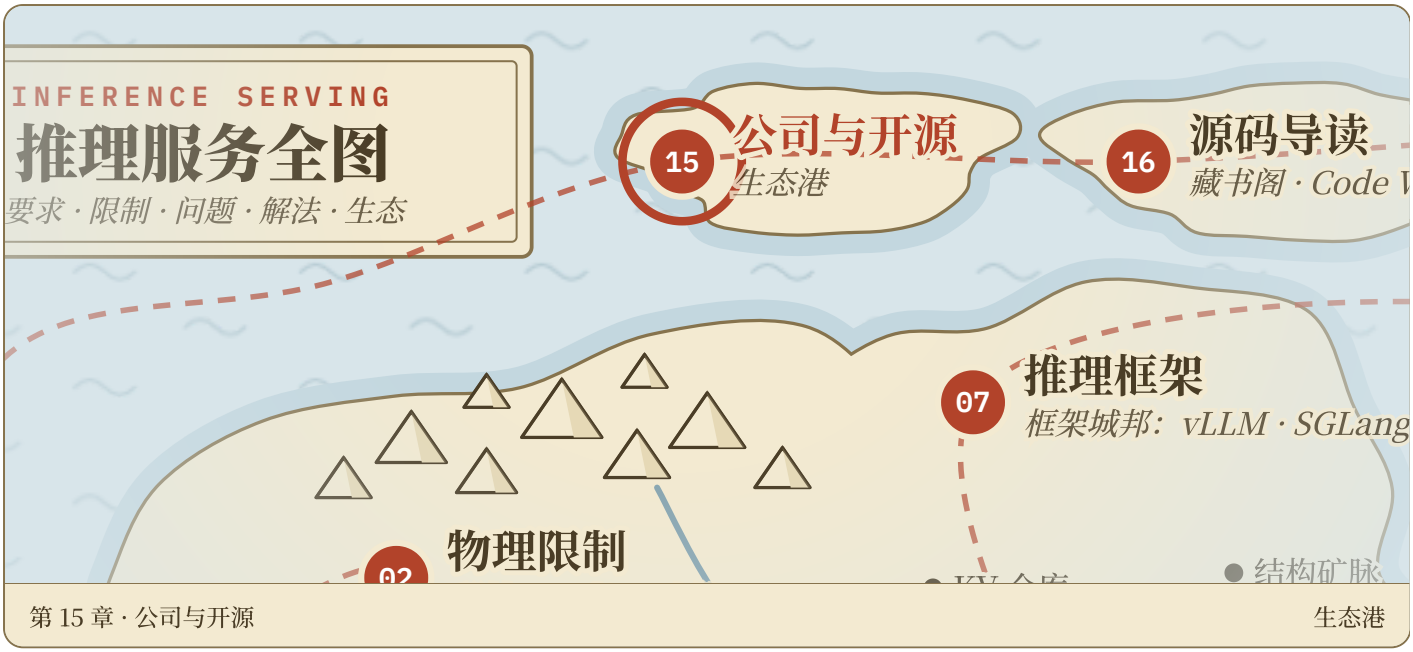
安全与合规

- 内容安全分类器（Llama Guard、ShieldGemma、NeMo Guardrails）与主模型并行或流式运行，避免串行增加延迟。
- Prompt injection 防护（尤其 agent 读取外部内容时）、PII 脱敏、日志留存策略。
- 机密计算（H100 / Blackwell 的 Confidential Computing 模式）用于敏感数据推理。

硬件与编译器生态（简表）

层	代表
GPU	NVIDIA Hopper / Blackwell（CUDA）、AMD Instinct MI300 / MI350（ROCm）、Intel Gaudi / Arc（XPU）、摩尔线程（MUSA）
云厂商自研芯片	Google TPU、AWS Trainium / Inferentia、Microsoft Maia、Meta MTIA

层	代表
推理专用架构	Groq LPU（SRAM 为主、确定性调度；2025 年底 NVIDIA 与其达成技术授权并吸纳核心团队）、Cerebras（晶圆级）、SambaNova（可重构数据流）、Etched（Transformer ASIC）
国内加速器	华为昇腾（CANN / MindIE，vllm-ascend）、寒武纪、海光、摩尔线程、沐曦等
Kernel 语言与库	CUDA、CUTLASS / CuTe DSL、Triton、TileLang、ThunderKittens；cuBLAS、FlashAttention、FlashInfer、DeepGEMM、NCCL / NVSHMEM
图编译	torch.compile（Inductor）、TensorRT、XLA（JAX / TPU）、TVM、MLIR 系



15 公司与开源项目地图

按技术栈从上到下分层。绿色描边是开源项目，其余是公司或商业服务。这个领域变化很快（融资、收购、更名频繁），下面反映的是约 2026 年中的格局，面试前建议再核对一次。

闭源模型 API
自研模型 + 自建推理

OpenAI · Anthropic · Google Gemini · xAI · Mistral · Cohere · DeepSeek · 阿里 通义 / 百炼 · 字节 豆包 / 火山方舟 · Moonshot Kimi · 智谱 Z.ai · MiniMax · 阶跃星辰 · 百度 千帆

开源模型推理平台
Inference-as-a-Service

Together AI · Fireworks AI · Baseten · DeepInfra · Groq · Cerebras · SambaNova · Nebius · Novita · Hyperbolic · SiliconFlow 硅基流动 · Replicate · fal · Modal · HF Inference Providers · Cloudflare Workers AI · OpenRouter（聚合）

云厂商托管
Model-as-a-Service

AWS Bedrock / SageMaker · Azure AI Foundry · Google Vertex AI · Oracle OCI · 阿里云 PAI / 百炼 · 火山引擎 · 腾讯云 · 百度智能云

GPU 云 / Neocloud
算力租赁

CoreWeave · Lambda · Nebius · Crusoe · RunPod · Vultr · Voltage Park · NVIDIA DGX Cloud Lepton

开源商业化公司
围绕开源项目

Inferact（vLLM，2026） · RadixArk（SGLang，2026） · Red Hat（llm-d、企业版 vLLM；2025 年收购 Neural Magic） · Anyscale（Ray） · Modular（MAX / Mojo） · Ollama · Hugging Face

集群编排 control plane	NVIDIA Dynamo · llm-d · AIBrix · KServe · Ray Serve · production-stack · Gateway API Inference Ext. · LWS
推理引擎 engine	vLLM · SGLang · TensorRT-LLM · LMDeploy · llama.cpp · Ollama · MLC-LLM · KTransformers · TGI
KV 存储与通信 data plane	Mooncake · LMCache · NIXL · DeepEP · 3FS · NCCL
Kernel 库 算子	FlashAttention · FlashInfer · FlashMLA · DeepGEMM · CUTLASS · Triton · TileLang · XGrammar
硬件 加速器	NVIDIA · AMD · Google TPU · AWS Trainium / Inferentia · Intel Gaudi · Groq · Cerebras · SambaNova · Etched · Tenstorrent · 华为昇腾 · 寒武纪 · 摩尔线程

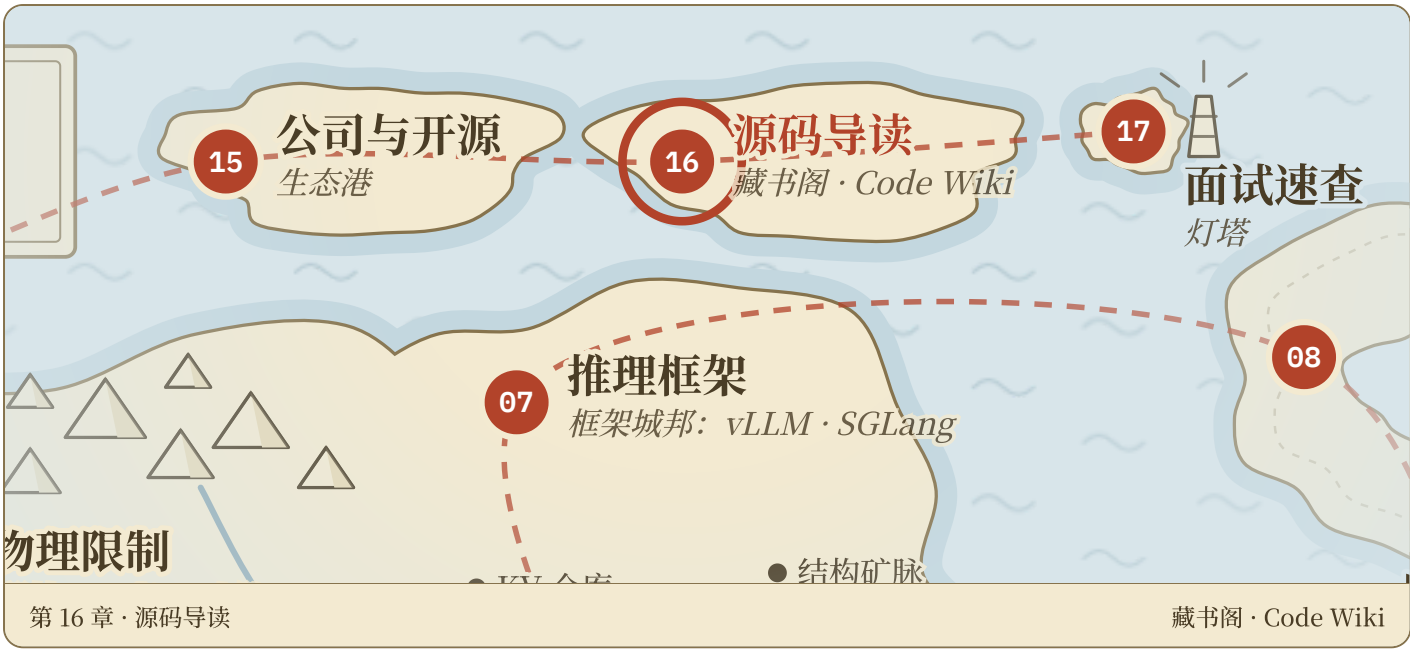
按领域

领域	公司 / 商业服务	开源项目
LLM 通用推理	见上方分层；NVIDIA NIM（预打包推理微服务）	vLLM、SGLang、TensorRT-LLM、LMDeploy、llama.cpp
MoE / 超大规模	DeepSeek、Moonshot、各大模型 API；NVIDIA（GB200 NVL72 方案）	DeepEP、DeepGEMM、EPLB、FlashMLA、Mooncake、Dynamo、llm-d
后训练 / RL	Thinking Machines（Tinker）、Prime Intellect、各推理平台的微调服务	verl、OpenRLHF、TRL、slime、Miles、AReaL、ROLL、NeMo-RL、SkyRL、prime-rl
语音 / Omni	ElevenLabs、Cartesia、Deepgram、AssemblyAI、Fish Audio、OpenAI Realtime、Gemini Live、Hume	vLLM-Omni、SGLang-Omni、faster-whisper、whisper.cpp、NVIDIA NeMo / Riva、CosyVoice、Fish Speech、Pipecat、LiveKit Agents
图像 / 视频生成	fal、Replicate、Runway、Luma、Pika、Black Forest Labs、Midjourney、快手可灵、MiniMax 海螺、生数 Vidu、Google Veo、OpenAI Sora	Diffusers、ComfyUI、xDiT、FastVideo、LightX2V、Nunchaku、Cache-DiT、SGLang Diffusion、vLLM-Omni
机器人	Physical Intelligence、Figure、Tesla Optimus、1X、Agility、Skild AI、Google DeepMind、NVIDIA、宇树、智元、银河通用	LeRobot、openpi、Isaac GR00T、Isaac Lab、OpenVLA、dora-rs、ROS 2

领域	公司 / 商业服务	开源项目
Coding agent	Anthropic（Claude Code）、OpenAI（Codex）、Cursor、GitHub Copilot、Windsurf、Cognition（Devin）、Replit；快速 apply 模型：Morph、Relace	OpenHands、SWE-agent / SWE-ReX、SWE-bench、Aider、OpenCode、Qwen Code、Gemini CLI
AI for Science	Google DeepMind / Isomorphic Labs（AlphaFold）、NVIDIA（BioNeMo、Earth-2）、EvolutionaryScale（ESM3）、Chai Discovery、Arc Institute（Evo 2）、Microsoft（Aurora）、Meta FAIR Chemistry（UMA）、ECMWF（AIFS）	AlphaFold3、OpenFold、Boltz、Chai-1、Protenix、ColabFold、ESM、Evo 2、GraphCast、Earth2Studio、MACE、e3nn、cuEquivariance、fairchem
Sandbox / agent 基础设施	E2B、Modal、Daytona、Runloop、Northflank、Cloudflare、Vercel	OpenSandbox（阿里开源）、Firecracker、gVisor、Kata Containers、OpenHands、SWE-ReX
网关 / 观测	OpenRouter、Portkey、Helicone、LangSmith、Arize	LiteLLM、Langfuse、Envoy AI Gateway、OpenTelemetry
端侧 / 本地	Apple、Qualcomm、LM Studio、Ollama	llama.cpp、MLX、MLC-LLM / WebLLM、ExecuTorch
基准 / 评测	SemiAnalysis（InferenceMAX）、Artificial Analysis、MLCommons	lm-evaluation-harness、OpenCompass、guidellm、AIPerf

近两年值得知道的行业事件

- 2025-01：Red Hat 完成收购 Neural Magic（vLLM 主要贡献者之一）；2025 年 llm-d 项目启动。
- 2025-02：DeepSeek Open Infra 周开源 FlashMLA、DeepEP、DeepGEMM、EPLB、3FS 等并公布推理系统数据。
- 2025-03：NVIDIA 在 GTC 开源 Dynamo。
- 2025-12：NVIDIA 与 Groq 达成非独家技术授权，Groq 创始人等加入 NVIDIA，Groq 继续独立运营 GroqCloud（报道称交易规模约 200 亿美元，未经官方确认）。
- 2026-01：vLLM 作者创立 Inferact，获 1.5 亿美元种子轮；SGLang 团队成立 RadixArk。



16 源码导读：带着问题读代码

文中所有源码链接都固定在某个 commit（与本文写作时 GitHub 上的 main 一致），行为与链接不会因为上游重构而失效；想看最新版本，在 GitHub 页面按 `y` 切换或手动换成 `main`。

◈ Code Wiki 链接指向 Google Code Wiki 为该仓库生成的结构化讲解（按模块解释架构和调用关系），适合在读源码前先建立全局图；若某个仓库还没生成，页面会提示。

怎么读最高效

- 先在 Code Wiki 上看模块划分，再按下面的“精读顺序”从入口一路跟到 kernel，每个文件只回答表里那一个问题。
- 用最小的模型（如 0.5B–1B）本地起服务，在 scheduler 的主循环和 model runner 的前向入口打断点，发一个请求，看一步里数据结构怎么变。
- 配合 PyTorch profiler / Nsight Systems 看一个 decode step 的时间线，把源码里的函数和时间线上的 kernel 对应起来。

LLM 引擎

主题	精读顺序	读的时候回答
vLLM 请求主路径	`api_server.py` `async_llm.py` `engine/core.py` `gpu_model_runner.py` ◈ Code Wiki	前端进程和 EngineCore 之间传的是什么？EngineCore 的忙循环里一步做了哪几件事？输出怎么流回 HTTP？
vLLM 调度与 KV	`scheduler.py` `kv_cache_manager.py` `block_pool.py` `kv_cache_utils.py`	一步的 token 预算如何在 running / waiting 请求之间分配？何时触发抢占？前缀命中如何计算（block 哈希链）？

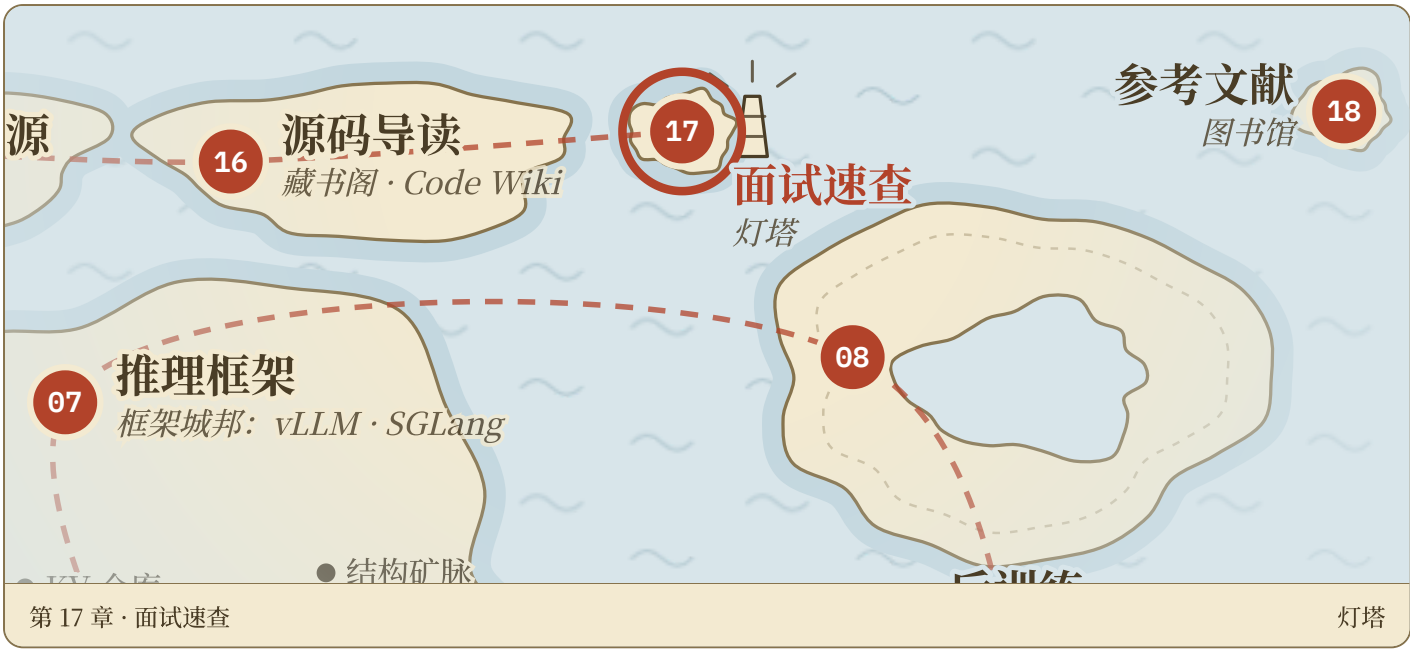
主题	精读顺序	读的时候回答
vLLM 执行层	`gpu_input_batch.py` `cudagraph_dispatcher.py` `compilation/backends.py` `flashinfer backend`	persistent batch 如何增量更新？哪些批大小走 CUDA graph？attention 的 metadata（block table、seq lens）在哪里构造？
SGLang 请求主路径	`http_server.py` `tokenizer_manager.py` `scheduler.py` `detokenizer_manager.py` `io_struct.py` `◈ Code Wiki`	三个进程之间通过 ZMQ 传哪些结构？scheduler 的事件循环（含 overlap 版本）如何组织？
SGLang 调度与缓存	`schedule_policy.py` `schedule_batch.py` `radix_cache.py` `memory_pool.py`	cache-aware 调度如何按前缀长度排序？radix tree 的插入、分裂、LRU 淘汰怎么实现？req_to_token 与 token_to_kv 两级映射是什么？
TensorRT-LLM	`py_executor.py` `resource_manager.py` `xqa kernels` `◈ Code Wiki`	PyTorch 后端的执行循环与 vLLM 有何异同？KV cache 资源如何管理？
llama.cpp	`llama-context.cpp` `llama-batch.cpp` `llama-kv-cache.cpp` `server/`	端侧引擎如何在不依赖 PyTorch 的情况下做批处理与 KV 管理？

特性专题

主题	读什么	读的时候回答
PD 分离	`KVConnector 接口` `NixlConnector` `SGLang prefill.py` `decode.py` `NIXL API`	connector 的 scheduler 侧和 worker 侧各负责什么？KV 是逐层还是整体传？decode 端如何预留空间并得知传输完成？
投机解码	`eagle.py` `rejection_sampler.py` `SGLang eagle_worker_v2.py` `EAGLE`	draft 树怎么构造？验证时 attention mask 怎么处理？被拒绝 token 的 KV 怎么回滚？
MoE / EP	`FusedMoE layer` `modular_kernel.py` `two_batch_overlap.py` `eplb.py` `DeepEP kernels` `◈ DeepEP · Code Wiki`	prepare / experts / finalize 三段如何解耦以便替换通信后端？TBO 怎么切微批？EPLB 如何决定冗余专家？
结构化输出	`XGrammar cpp/` `vLLM backend` `SGLang backend`	哪些 token 可以预计算（上下文无关）？bitmask 何时生成、在哪个线程、如何与 GPU 前向重叠？

主题	读什么	读的时候回答
集群路由	`Dynamo kv_router` `llm-d EPP` `SGLang Model Gateway` `◈ Dynamo · Code Wiki`	路由器如何得知每个实例缓存了哪些 block（KV 事件 vs 近似树）？打分函数如何平衡命中与负载？
RL 集成	`verl rollout/` `checkpoint_engine` `slime backends` `SGLang weight_sync` `batch_invariant.py`	权重从训练布局到推理布局如何 reshard 与传输？引擎如何让出显存？确定性模式替换了哪些 kernel？
Omni 多阶段	`vLLM-Omni engine` `OmniConnector` `SGLang-Omni pipeline` `comm/`	一个请求如何在阶段间流转？chunk 何时交给下一阶段？每个阶段的调度器有何不同？
扩散	`Diffusers pipelines` `xDiT core` `cache-dit caching` `vLLM-Omni 扩散调度`	去噪循环在哪里？USP 如何切分序列与 head？缓存命中的判定阈值如何计算？步级批处理如何对齐不同请求？
Kernel	`FA2 fwd kernel` `FA3` `FlashInfer decode` `DeepGEMM` `custom all-reduce` `CuTe 教程`	online softmax 的 rescale 在哪一行？warp specialization 的生产者 / 消费者如何同步？one-shot all-reduce 如何用 IPC 读对端显存？
Sandbox	`OpenSandbox execd` `OpenSandbox server` `Firecracker snapshot` `gVisor runsc` `◈ OpenSandbox · Code Wiki`	sandbox 的生命周期（创建、执行、暂停、销毁）由谁管理？命令执行与文件操作的 API 如何设计？快照恢复如何缩短冷启动？
机器人	`openpi policy server` `LeRobot policy server` `GR00T deployment`	观测如何打包传给服务端？动作块如何返回与执行？
AI for Science	`AF3 evoformer` `Boltz 三角注意力` `cuEquivariance triangle` `MACE blocks`	成对表示在哪里分块（chunk size）？三角注意力的 bias 怎么加？等变张量积如何被拆成可融合的 kernel？

更多 Code Wiki `◈ LMCache` `◈ Mooncake` `◈ llm-d` `◈ XGrammar` `◈ slime` `◈ AReaL` `◈ FastVideo` `◈ Isaac GR00T` `◈ E2B`



第 17 章 · 面试速查　　灯塔

17 面试速查

估算公式（背下来，现场能算）

```
权重显存        = N_params × bytes（BF16=2, FP8=1, FP4=0.5）          70B BF16 ≈ 140 GB
KV / token      = 2 × L × n_kv × d_head × bytes                       Llama-3-70B ≈ 32
最大 KV token   ≈（显存 × util − 权重 − 激活/图）/ KV per token
decode 下限     ≈（权重字节 + 本批 KV 字节）/ HBM 带宽                 8B@H100 ≈ 4.8 ms/
prefill 时间    ≈ 2 × N_params × prompt_tokens /（峰值 × MFU）         8B, 2k tok, MFU
并发            = 到达率 × 平均停留时间（Little）
投机期望 token  =（1 − α^(k+1)）/（1 − α）
TP all-reduce   每层 2 次，每次 ≈ tokens × hidden × bytes
成本 / 1M tok   = GPU 时价 /（tok/s/GPU × 3600 / 1e6）                $2.5/h, 2500 tok
```

例题：Llama-3-70B（BF16）放在 8×H100 上，TP=8，能装多少 KV？

权重 140 GB ÷ 8 = 17.5 GB/卡；按 0.9 利用率可用 72 GB，扣掉权重和约 4 GB 激活 / CUDA graph，每卡约 50 GB 给 KV，8 卡共约 400 GB。GQA 有 8 个 KV head，TP=8 时每卡正好 1 个 head，每 token 的 320 KiB 均摊到 8 卡。总容量 ≈ 400 GB ÷ 320 KiB ≈ 120 万 token，即约 300 个 4k 上下文的并发请求。

高频问题

▼ 为什么 decode 是 memory-bound？怎么提高它的硬件利用率？

每个 decode step 每个请求只处理 1 个 token，却要把全部权重从 HBM 读一遍，算术强度约等于批大小 B，远低于 H100 约 295 FLOP/byte 的脊点；attention 读 KV 的强度约为 1，且不能被批摊薄。

- 加大批：continuous batching + PagedAttention 腾出 KV 空间 + 量化 / MLA 缩小 KV。
- 减少字节：权重 FP8 / FP4、KV FP8。
- 一次算多个 token：投机解码、MTP。
- 减少 CPU 空泡：CUDA graph、重叠调度。

▼ 解释 PagedAttention，它的代价是什么？

KV 按固定大小 block 分页，block table 把逻辑块映射到物理块，按需分配，消除预留浪费和外部碎片，并支持引用计数共享与 copy-on-write。代价：attention kernel 需要间接寻址（早期比连续布局慢一些，现在 FlashInfer / FA 已基本抹平）；block 太大有内部碎片，太小则元数据和寻址开销上升；所有 kernel（attention、KV 写入、拷贝）都要支持分页布局。

▼ vLLM 的 prefix caching 和 SGLang 的 RadixAttention 有什么不同？

vLLM 用“父哈希 + 本块 token”的链式哈希给每个满 block 编号，查表 O(1)，只能以 block 为粒度命中；SGLang 用 radix tree 存 token 序列，可以 token 粒度匹配，天然表达树状共享，并配合 cache-aware 调度和 LRU 叶子淘汰。两者效果在常见负载上接近，radix tree 在复杂分支（多轮、树搜索、多采样）上更直观。

▼ Chunked prefill 和 PD 分离，什么时候用哪个？

两者都解决 prefill 干扰 decode。chunked prefill 在同一 GPU 内按时间切分，零额外基础设施，中小规模首选；代价是 prefill 效率略降、两阶段仍共享并行配置。PD 分离在空间上隔离，适合大规模、长 prompt、TTFT 和 TPOT SLO 都严格、或两阶段需要不同并行度（如 DeepSeek 的 EP32 vs EP144）的场景；代价是 KV 传输（需要 RDMA）和系统复杂度。

▼ 投机解码为什么能保证输出分布不变？什么时候不该用？

验证时对每个 draft token 以 $\min(1, p_{target}/p_{draft})$ 的概率接受，拒绝时从 $(p_{target} - p_{draft})_+$ 归一化后的分布重新采样，可以证明最终每个位置的分布与目标模型直接采样完全相同（贪心时就是逐 token 比对）。高并发（GPU 已 compute-bound）、接受率低（高温、开放创作）、draft 成本高时收益为负。

▼ MoE 推理的主要挑战？为什么 DeepSeek 用 DP attention + 大 EP?

- 显存按总参数算，计算按激活参数算；decode 时每个专家分到的 token 很少，GEMM 小而 memory-bound。
- EP 带来每层两次 all-to-all，且负载不均。
- 宽 EP 让每卡只放少数专家，显存留给 KV，全局 token 汇聚后每专家批变大；MLA 不适合 TP，所以 attention 用 DP。配套 DeepEP、TBO 重叠、EPLB 冗余专家、DeepGEMM grouped GEMM。

▼ TP、PP、EP 怎么选?

先看能否放下：单卡放得下就用 DP 多副本。放不下时机内优先 TP（降延迟，需要 NVLink）；跨机时 TP 通信太贵，用 PP（不降单 token 延迟）或对 MoE 用 EP。TP 太大时每卡 GEMM 变瘦、通信占比上升，收益递减。长 prompt prefill 可加 CP。MoE 的常见组合是 attention DP（或小 TP）+ 专家 EP。

▼ 线上 TTFT p99 突然变差，怎么排查?

- 拆解 TTFT：排队时间 vs prefill 时间（看引擎的 queue time 指标）。
- 排队变长：流量上涨、扩容滞后、某实例被路由打爆（看各实例负载是否均衡）、抢占重算增多（KV 不足）。
- prefill 变长：长 prompt 比例上升、前缀命中率下降（路由策略或会话粘性变化、cache 被挤出）、chunk 设置变化。
- CPU 侧：tokenizer / 多模态预处理成为瓶颈、API server 进程不够。
- 最后看硬件：降频、NVLink / 网卡异常、某张卡 ECC 错误。

▼ 设计一个支持大量用户的聊天服务（system design 骨架）

- 需求：模型规模、QPS、ISL/OSL 分布、TTFT / TPOT SLO、成本目标。
- 容量估算：用 Little 定律算并发 → 算 KV 需求 → 每实例并发上限 → 实例数；用基准曲线选满足 SLO 的批大小和每 GPU 吞吐。
- 单实例：引擎选型、并行方式、量化、chunked prefill、前缀 cache、投机。
- 集群：API 网关（鉴权、限流）→ KV 感知路由 →（可选 PD 分离）→ 分层 KV 池；按队列 / SLO 指标自动扩缩；多区域。
- 可靠性：健康检查、重试、优雅下线、降级（换小模型、限制 max_tokens）。
- 观测：TTFT / ITL 分位数、命中率、KV 使用率、错误率、成本看板。

▼ 量化模型上线前怎么验证?

先确认 kernel 路径（是否真的走了 FP8 / FP4 Tensor Core，还是退回反量化）；精度上在目标任务上做回归（数学、代码、长上下文检索、多语言），不要只看 perplexity；用相同采样参数对比输出分布；压测确认吞吐和延迟收益；灰度上线并监控用户侧指标。

▼ RL rollout 和在线 serving 有什么不同?

目标是 step 时间而不是单请求 SLO；权重每步更新（共置用 CUDA IPC + sleep mode，分离用 NCCL / RDMA 广播）；长尾严重（partial rollout、异步 RL、动态采样）；要求推理 logprob 与训练一致（重算 + 重要性采样修正、MoE 路由回放、确定性 kernel）；agentic 任务需要大量 sandbox 和多轮 KV 复用。

▼ Omni / 语音模型的 serving 和 LLM 有什么不同?

一个模型是多个异构阶段（encoder、Thinker、Talker、codec 解码器），各需不同的批处理和资源；阶段间要传张量；输出是实时流，核心指标变成首包延迟和 RTF，要 chunk 级异步交接；全双工需要持久会话和按节拍推进的调度；多模态预处理和 encoder 可以缓存或单独拆分（EPD）。

▼ 扩散模型推理怎么加速?

它是 compute-bound 的固定步数迭代：减步数（蒸馏）收益最大；免训练的特征缓存（TeaCache、FBCache 等）复用相邻步特征；视频长序列用序列并行（Ulysses / Ring / USP）和稀疏 attention；CFG 并行；FP8 / 4-bit 量化；torch.compile / TensorRT；VAE 分块与并行。serving 上用异步任务 API、步级批处理、LoRA 热切换。

▼ 机器人策略推理为什么不能直接套 LLM serving?

它是闭环实时控制：batch = 1、看最坏延迟和抖动、多速率（VLM 5–10 Hz、动作 50–200 Hz、底层控制 1 kHz）、多在边缘设备上跑、错过周期有安全后果。核心技术是 action chunking、异步推理 + RTC 平滑拼接、快慢双系统、量化编译与 CUDA graph，以及安全监控和回退。

▼ Coding agent 的推理负载有什么特殊？怎么优化?

模型是长上下文、会调用工具、带 thinking 的大模型（多为 MoE），经过 SWE 类 agentic RL 训练；会话是几十上百轮的“追加式”上下文，每轮只追加工具输出，前缀几乎全部可复用，轮与轮之间要等工具执行几秒到几分钟。所以：前缀命中率是核心指标；KV 要跨工具空档保留（GPU LRU + CPU / SSD 分层）；路由要会话粘性或 KV 感知；长 prompt 用 chunked prefill 隔离；TPOT 决定 agent 完成时间；tool parser 必须可靠；sandbox 的创建与快照恢复在关键路径上。行内补全则相反：小模型、极低延迟、高 QPS、大量取消；apply / 编辑用原文件做投机 draft。

▼ AlphaFold 类模型的推理瓶颈在哪里？和 LLM 有什么不同?

成对表示是 $N \times N \times c$，三角注意力接近 $O(N^3)$，显存随残基数平方增长，几千残基就要分块甚至 offload；每个复合物单独推理（batch = 1、形状各异）。MSA 搜索在 CPU 上扫 TB 级数据库，常比 GPU 推理还慢，所以要 MMseqs2 加速和缓存。AF3 还有扩散模块和多 seed 采样。精度上以 BF16 / TF32 为主。优化手段：分块、三角注意力专用 kernel（cuEquivariance 等）、MSA 服务化与缓存、按大小分桶调度。

▼ 为什么分子动力学里的 ML 势函数（MLIP）推理更看重延迟而不是吞吐?

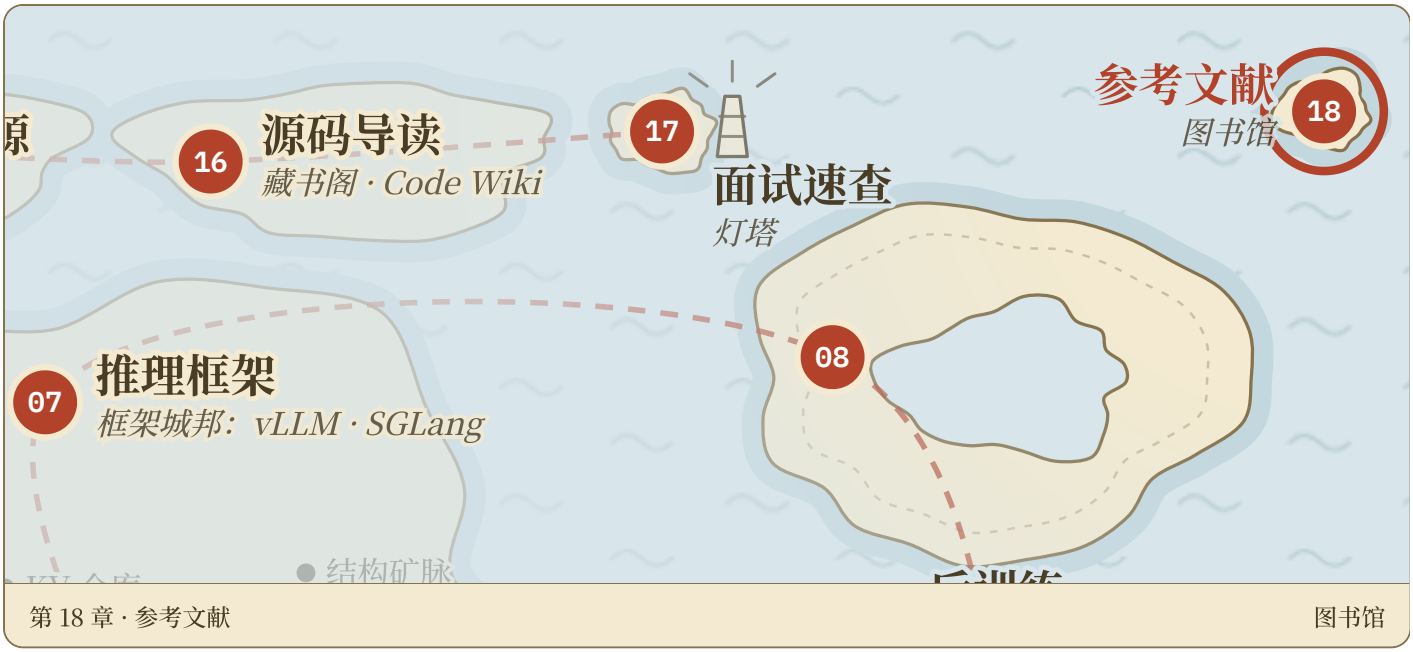
MLIP 被嵌在 MD 积分循环里，每个时间步都要算一次能量和力，模拟要跑百万步，而每步只是一个几百到几千原子的小图。总时间 = 步数 × 单步延迟，单步又很小，kernel launch 和小算子开销占主导；等变网络的张量积是计算热点。所以要 CUDA graph、融合的等变 kernel，并保证 FP32 级别的数值稳定性（能量守恒）。

▼ nvidia-smi 显示 GPU 利用率 100%，为什么吞吐还是很低?

nvidia-smi 的 utilization 只表示“采样窗口内有 kernel 在跑的时间比例”，一个只占用几个 SM 的 memory-bound kernel 也会显示 100%。应该看 MFU / MBU、SM 占用率、Tensor Core 活跃度（DCGM 的 profiling 指标或 Nsight），并检查批大小是否太小、是否有 CPU 空泡。

▼ 什么是 goodput？为什么比 throughput 更合适?

goodput 只统计满足 SLO（如 TTFT < 1 s 且 TPOT < 50 ms）的请求所贡献的吞吐。只看 throughput 时，把批无限加大就能刷高数字，但大量请求已经违约，对业务没有价值。goodput 把延迟约束纳入优化目标，是 DistServe 等工作用来评估 PD 分离的口径。



18 参考文献与延伸阅读

LLM serving 系统

- Orca (OSDI’22) — iteration-level scheduling
- vLLM / PagedAttention (SOSP’23)
- SGLang / RadixAttention
- Sarathi-Serve (OSDI’24) — chunked prefill
- DistServe (OSDI’24) · Splitwise · Mooncake
- Llumnix — 请求迁移
- S-LoRA · Punica
- XGrammar

Kernel、量化、结构

- FlashAttention · FA2 · FA3
- FlashInfer
- GPTQ · AWQ · SmoothQuant
- GQA · DeepSeek-V2（MLA） · DeepSeek-V3
- Defeating Nondeterminism in LLM Inference

投机解码与并行

- Leviathan et al. · Chen et al.
- Medusa · EAGLE · EAGLE-2 · EAGLE-3
- Megatron-LM · Ring Attention · DeepSpeed-Ulysses
- SGLang 大规模 EP

后训练 / RL

- HybridFlow（verl）
- OpenRLHF
- AReaL — 异步 RL
- DeepSeekMath（GRPO）

Omni 与扩散

- Qwen2.5-Omni · Qwen3-Omni · Moshi
- vLLM-Omni
- DiT · Wan
- xDiT · USP · DistriFusion · PipeFusion
- DeepCache · TeaCache · SVDQuant
- LCM · DMD2 · Sliding Tile Attention

机器人

- RT-2 · OpenVLA
- π0 · π0.5 · FAST
- GR00T N1 · Gemini Robotics
- ACT（action chunking） · Real-Time Chunking · SmolVLA

Coding agent 与 AI for Science

- SWE-bench · SWE-agent
- AlphaFold2 (Nature 2021) · AlphaFold3 (Nature 2024)
- GraphCast · MACE

Sandbox 与基础设施

- Firecracker (NSDI’20)
- gVisor · OpenSandbox
- Google Code Wiki：为 GitHub 仓库生成结构化讲解
- DeepSeek Open Infra Index
- Roofline model · Little’s law

文档

- vLLM 文档 · SGLang 文档
- TensorRT-LLM 文档 · Dynamo 文档
- llm-d · vLLM-Omni 文档 · SGLang-Omni 文档