5. 常见模型

Table of contents

1 概览：为什么模型架构重要 **2**

2 架构原型：四种基本骨架 **3**

3 数学与机制速览 **4**
- 3.1 注意力 . . . 4
- 3.2 KV 缓存：为什么 decoder-only 会越聊越贵 . . . 5
- 3.3 MHA、MQA 与 GQA：为什么注意力头设计会决定服务成本 . . . 6
- 3.4 MLP / FFN：为什么注意力之外还要有一块“局部计算” . . . 7
- 3.5 三种常见训练目标 . . . 7
- 3.6 位置编码：上下文长度为什么不是“白送”的 . . . 8
- 3.7 MoE 路由：为什么它能扩容量，也能带来系统麻烦 . . . 8

4 Encoder-Only 模型 **9**
- 4.1 BERT . . . 9

5 Decoder-Only 模型 **10**
- 5.1 为什么它主导了现代 LLM . . . 10
- 5.2 GPT 家族：统一生成接口的胜利 . . . 10
- 5.3 PaLM：大规模 decoder 的系统化配方 . . . 11
- 5.4 LLaMA：现代开源 decoder 配方的母体 . . . 11
- 5.5 Qwen：同骨架下，数据与后训练如何拉开差距 . . . 11
- 5.6 Mistral：效率优先的 decoder 设计 . . . 11
- 5.7 Decoder-only 为什么仍然不是“默认万能答案” . . . 12

6 Encoder-Decoder 与 Hybrid 模型 **12**
- 6.1 T5：当任务天然是“把一个输入变成另一个输出” . . . 12
- 6.2 PrefixLM / GLM：介于理解与生成之间的折中 . . . 13

7 Dense 模型 vs Mixture-of-Experts (MoE) **13**

8 Long-Context 架构 **15**

9 多模态架构 **18**

10 如何评估模型架构 **19**
10.1 能力基准 20
10.1.1 效率指标 20
10.1.2 任务贴合度 20
10.1.3 评测方法 20
10.2 常见评测陷阱 21
10.3 工具、排行榜与第一手资料 21
10.4 推荐的最小基准集合 22

11 架构究竟改变了什么 **25**

12 工程案例 **26**
12.1 把聊天模型拿去做 reranker，质量没上去，成本先爆了 26
12.2 MoE 模型离线很强，线上尾延迟却失控 26
12.3 128K 写在 model card 上，系统却在 128K 里死掉了 26
12.4 公开榜单前排模型，客服工单却解不掉 27
12.5 多模态 demo 很惊艳，文档理解却不可靠 27

13 本章小结 **27**
13.1 问题小结 29
13.2 给 AI 工程师的实践清单 30

14 参考资料 **31**

1 概览：为什么模型架构重要

想象一个企业搜索团队：他们看到公开榜单上某个 70B 聊天模型很强，于是直接拿它做文档排序。结果并不戏剧化，但足够致命：相关性没有明显提升，延迟却上去了，GPU 成本放大了，批量吞吐掉了，线上系统还更难调试。后来他们把排序层改成 encoder-only reranker，把 decoder-only 模型只留给最终答案综合，质量反而更稳，成本也显著下降。问题不在调参，而在架构匹配。

这就是本章的起点。模型架构不是“论文里的名字”，也不是“排行榜上的品牌”。它决定的是：

- 模型更擅长理解还是生成；
- 训练成本是更受参数量、路由还是长序列影响；
- 推理延迟是被串行解码、KV 缓存还是专家通信主导；
- 长上下文是可营销的规格，还是可服务的能力；
- 多模态输入是被稳健地对齐，还是只在 demo 中看起来工作正常。

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 主流语言模型究竟可以归结为哪几类稳定架构原型？
> 2. 为什么很多模型名称不同，底层却共享相似的工程配方？
> 3. 为什么评估模型时，排行榜分数绝不能替代架构理解与场景验证？

从工程视角，选架构本质上是一个受约束优化问题。你可以把它写成：

$$\mathcal{J}(\mathcal{A}) = \lambda_q \cdot \mathrm{Err}(\mathcal{A}) + \lambda_\ell \cdot \mathrm{Latency}(\mathcal{A}) + \lambda_c \cdot \mathrm{Cost}(\mathcal{A}) + \lambda_r \cdot \mathrm{Risk}(\mathcal{A})$$

其中 $\mathcal{A}$ 是架构选择，Err 是任务误差，Latency 是端到端延迟，Cost 是训练与推理成本，Risk 是系统失控、故障和安全风险。不同架构并不是在同一个目标面上平移，它们会改变整个代价函数的形状。

因此，本章要回答的不是“哪个模型最好”，而是一个更稳定、也更接近生产的问题：

> 不同的模型架构，究竟如何塑造 **AI** 系统的能力、成本与故障模式？

这一章讨论的不是“哪个模型最好”，而是主流模型到底在架构上如何分化、为什么会分化，以及这些差异在工程上意味着什么。很多模型名称看起来像一串品牌，但如果把营销名称剥掉，底层仍然可以归结为几类稳定的原型：encoder-only、decoder-only、encoder-decoder、PrefixLM，以及在这些骨架之上叠加的 MoE、长上下文、局部注意力、多模态等变化。

本章还会回答另一个工程上更重要的问题：我们该如何评估这些模型，而不是只会复述排行榜。模型选择不是看单一分数，而是看架构、训练目标、推理成本、评测口径和目标产品场景之间是否一致。

读完这一章，你应该能够：

- 解释几类主要 Transformer 原型（encoder-only、decoder-only、encoder-decoder、PrefixLM）及其适用场景。
- 读懂注意力、训练目标、KV 缓存、RoPE、MoE 路由等核心公式，并知道它们在工程上意味着什么。
- 理解推理阶段最常见的权衡：MHA / MQA / GQA、KV 缓存规模、上下文长度与延迟之间的关系。
- 从技术报告、model card 和配置文件中找到并验证模型的“真实规格”。
- 设计一套小而有效的评估集合，并避开最常见的评测陷阱。

2 架构原型：四种基本骨架

! Important

本节先回答几个关键问题：

1. encoder-only、decoder-only、encoder-decoder 和 PrefixLM 的结构差别到底是什么？
2. 为什么今天的通用聊天模型几乎都围绕 decoder-only 收敛？
3. 什么时候你应该刻意选择 BERT 或 T5，而不是默认上聊天 LLM？

如果把品牌名全部剥掉，主流语言模型大致可以归结为下面四类原型。它们的区别，不在“是不是 Transformer”，而在信息如何流动。

- **Encoder-only** 更像表示学习器，适合分类、检索、排序、嵌入；
- **Decoder-only** 更像通用生成器，适合开放式生成、对话、代码和工具调用；
- **Encoder-decoder** 更像条件生成器，适合翻译、摘要、改写与结构化转换；

- **PrefixLM** 试图把“充分读取上下文”和“保持生成一致性”放进同一个掩码框架。

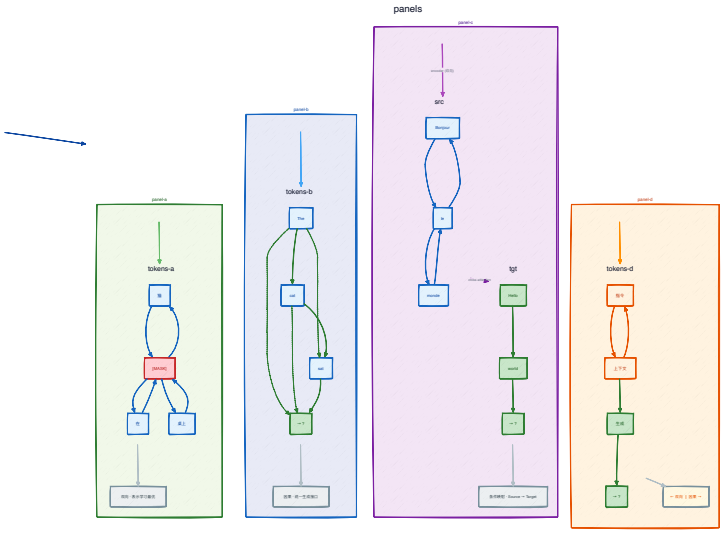

Figure 1: 不同架构的信息流对比：同是 Transformer，信息流方向决定一切

为什么今天的大多数通用 LLM 选择 decoder-only？因为它把训练目标、数据组织、对齐路径和推理接口全部统一成了“给定前缀，继续生成”。这种统一极其有利于大规模预训练与产品化。但这并不意味着其他架构消失了。如果任务是检索和排序，encoder 风格仍常常更合适；如果任务是翻译与摘要，seq2seq 风格仍然非常合适。

3 数学与机制速览

> **!** Important
>
> 在比较模型家族之前，先固定几条决定生产成本的机制：注意力、KV 缓存、位置编码、MoE 路由和多模态接口。真正的系统代价，大多从这里长出来。

这一节不是为了把公式背下来，而是为了建立一个足够稳固的工程直觉：为什么有些模型更贵、为什么有些模型更适合生成、为什么长上下文和 **MoE** 会迅速把系统复杂度推高。

3.1 注意力

Transformer 的核心不是“更深的网络”，而是把 token 之间的信息交换显式写成一层动态路由。给定输入表示 $X \in \mathbb{R}^{n \times d}$，一个标准注意力头写成：

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V$$

其中 M 是 mask。对 decoder-only 模型来说，最常见的是因果掩码：当前位置只能读取自己和左侧 token，不能看到未来 token。没有这层约束，自回归生成的训练目标就会失效。

为什么要除以 $\sqrt{d_k}$？因为高维点积的方差会随维度上升，不缩放时 softmax 更容易进入过尖区域，梯度更差，训练更不稳定。这个缩放不改变“谁更相关”的相对排序，但会改变数值条件。

工程上真正昂贵的是复杂度。标准全局自注意力要形成一个 $n \times n$ 的分数矩阵，因此：

- 时间 / 显存大致按 $O(n^2)$ 增长；
- 若隐藏维度为 d，整体计算常近似写成 $O(n^2 d)$；
- 若直接物化分数矩阵，单头中间张量大小就是 n^2 个元素。

一个经常被忽略的数量级是：当 $n = 131{,}072$（即 128K）时，单个头的注意力分数矩阵就有

$$n^2 = 131072^2 = 17{,}179{,}869{,}184$$

个元素。若以 bf16 存储，每个元素 2 字节，那么仅单头分数矩阵就约为 32 GiB。现实系统不会这样物化，这正是 FlashAttention 一类内核优化从“加速技巧”变成“必要条件”的原因。

3.2 KV 缓存：为什么 decoder-only 会越聊越贵

在 decoder-only 推理里，模型一次只生成一个新 token。第 t 步生成时，前 $t-1$ 个 token 的 K 和 V 不会改变；如果每一步都重新计算历史 token 的键值表示，代价会高得难以服务在线请求。因此推理系统会把历史的 K、V 存起来，也就是所谓 **KV cache**。

单层缓存大小可以粗略写成：

$$\text{KV bytes} \approx 2 \cdot B \cdot n \cdot G \cdot d_k \cdot b$$

其中：

- B 是 batch size；
- n 是当前序列长度；
- G 是 KV 组数；
- d_k 是每个 KV 头的维度；
- b 是每个元素的字节数；
- 前面的 2 来自同时存储 K 和 V。

若模型有 L 层，则总缓存近似为：

$$\text{KV bytes}_{\text{total}} \approx 2 \cdot B \cdot n \cdot G \cdot d_k \cdot b \cdot L$$

** 一个数值例子：32K 与 128K 在服务上差多少

假设一个 decoder-only 模型具备如下配置：

- $L = 80$ 层；
- $G = 8$ 个 KV 组；
- $d_k = 128$；
- bf16，因此 $b = 2$ 字节；
- 先看单请求，即 $B = 1$。

当 $n = 32{,}768$（32K）时：

$$\mathrm{KV}_{\mathrm{layer}} = 2 \cdot 1 \cdot 32768 \cdot 8 \cdot 128 \cdot 2 = 134{,}217{,}728 \text{ bytes} \approx 128 \text{ MiB}$$

乘上 80 层，总 KV 缓存约为：

$$128 \text{ MiB} \times 80 \approx 10 \text{ GiB}$$

当 $n = 131{,}072$（128K）时：

$$\mathrm{KV}_{\mathrm{layer}} = 2 \cdot 1 \cdot 131072 \cdot 8 \cdot 128 \cdot 2 = 536{,}870{,}912 \text{ bytes} \approx 512 \text{ MiB}$$

乘上 80 层，总 KV 缓存约为：

$$512 \text{ MiB} \times 80 \approx 40 \text{ GiB}$$

这还只是 **batch size = 1**。若 $B = 8$，则仅 KV 缓存理论上就会逼近 320 GiB。也就是说，很多“支持 128K”的模型并不是算不出，而是一旦走到真实并发，显存先死。

3.3 MHA、MQA 与 GQA：为什么注意力头设计会决定服务成本

设查询头数为 H，KV 组数为 G：

- **MHA**：$G = H$。每个查询头都有独立的 K, V，表达能力强，但缓存最大。
- **MQA**：$G = 1$。所有查询头共享一套 K, V，缓存最省，解码最轻，但可能损失部分质量。
- **GQA**：$1 < G < H$。多个查询头共享一组 K, V，是质量和缓存之间的折中。

KV 缓存大致正比于 G。因此从 MHA 到 GQA，再到 MQA，最直接的收益不是“参数更少”，而是解码阶段每个请求占用的缓存更小，系统更容易支撑长上下文和更高并发。

在服务系统里，可以把单 token 解码的带宽压力粗略理解为：

$$T_{\mathrm{decode}} \propto \frac{L \cdot n \cdot G \cdot d_k}{\text{HBM bandwidth}}$$

因此，只要 n 很长，哪怕 FLOPs 没有失控，**HBM** 带宽也会先成为瓶颈。这也是 GQA / MQA 在现实系统里如此重要的原因。

3.4 MLP / FFN：为什么注意力之外还要有一块“局部计算”

注意力负责在 token 之间路由信息；FFN 负责在每个 token 位置上做非线性变换。二者分工不同，缺一不可。没有 FFN，模型虽然还能交换信息，但表达能力会明显不足。

经典 FFN 写成：

$$\mathrm{FFN}(x) = \sigma(xW_1 + b_1)W_2 + b_2$$

现代 LLM 常见的是 GLU / SwiGLU 家族：

$$\mathrm{SwiGLU}(x) = (xW_a) \odot \mathrm{swish}(xW_b)$$

它比单路激活多一个门控路径，可以把一部分特征作为内容，另一部分作为调制信号。实践上，SwiGLU 往往能在相近预算下提供更好的表达效率，因此大量现代 decoder 都收敛到它。

从系统角度看，FFN 还有一个更现实的意义：它通常占了 **Transformer** 中很大一部分参数和 **FLOPs**。这也是为什么很多 MoE 优先把 FFN 稀疏化，而不是先把注意力层做成专家。

3.5 三种常见训练目标

训练目标会塑造模型更擅长哪类任务，也会反过来约束架构选择。

- **MLM (masked language modeling)**：遮掉部分 token，让模型利用左右文恢复它们。
- **Causal LM**：按从左到右的顺序预测下一个 token，即 $p(x_t \mid x_{<t})$。
- **Seq2Seq**：先编码输入 x，再条件生成输出 y。

对应损失常写成：

$$\mathcal{L}_{\mathrm{CLM}} = -\sum_{t=1}^{n} \log p(x_t \mid x_{<t})$$

$$\mathcal{L}_{\mathrm{MLM}} = -\sum_{t \in \mathcal{M}} \log p(x_t \mid x_{\setminus \mathcal{M}})$$

$$\mathcal{L}_{\mathrm{S2S}} = -\sum_{t=1}^{m} \log p(y_t \mid y_{<t}, x)$$

为什么现代通用 LLM 大多采用 Causal LM？因为它把训练目标和推理形式统一成了同一件事：给定前缀，继续生成。这让预训练、继续预训练、指令微调和推理接口都沿同一条工程路径扩展。

但这并不意味着 MLM 和 Seq2Seq 过时。前者仍非常适合表示学习、检索和分类；后者仍是翻译、摘要、改写、结构化抽取等条件生成任务的自然选择。

3.6 位置编码：上下文长度为什么不是“白送”的

Transformer 本身并不知道 token 顺序，因此必须显式引入位置信息。RoPE（旋转位置嵌入）是现代 decoder 中非常常见的选择，它对 Q 和 K 施加与位置有关的旋转：

$$\mathrm{RoPE}(q_t) = R(t)q_t, \quad \mathrm{RoPE}(k_t) = R(t)k_t$$

直觉上，RoPE 不是把位置编号直接加到向量里，而是让不同位置的表示在多个频率上发生相对旋转，于是注意力分数天然携带相对位置信息。

问题在于，位置方案都有边界。所谓长度外推，就是让模型在比训练时更长的上下文上继续工作。数学上能算，不代表语义上还能稳定。训练时没见过那样的距离分布，推理时质量往往会退化。用户看到的是“支持 128K”，工程师看到的则应该是“在 128K 上是否仍能检索、引用和推理”。

3.7 MoE 路由：为什么它能扩容量，也能带来系统麻烦

MoE (Mixture-of-Experts) 的核心思想是：不是每个 token 都走完整个大 MLP，而是由一个路由器挑选少数几个专家激活。常见写法是：

$$g(x) = \mathrm{softmax}(Wx), \qquad \mathrm{MoE}(x) = \sum_{e \in \mathrm{TopK}(g(x))} g_e(x) E_e(x)$$

这样做的好处是，总参数可以很大，但单个 token 实际只激活其中一小部分，因此单 token FLOPs 不必随着总参数线性上涨。

但 MoE 还有两个工程上必须关心的量。

第一是专家容量。若一批 token 数量为 N，专家数为 E，每个 token 选 top-k 个专家，容量因子为 c，则单专家可承载的近似容量可写为：

$$C = \left\lceil c \cdot \frac{Nk}{E} \right\rceil$$

若某些专家被过多 token 选中，超过容量的 token 就要被丢弃、降级或重路由。

第二是负载均衡损失。一种常见的写法是引入辅助项，使每个专家接到的 token 比例 f_e 与平均路由概率 p_e 更接近均匀：

$$\mathcal{L}_{\mathrm{balance}} = \alpha E \sum_{e=1}^{E} f_e p_e$$

这里最重要的不是具体常数，而是工程事实：MoE 的问题不只在“有没有专家”，更在“专家会不会被稳定、均匀地使用”。

问题在于，MoE 的代价不是消失，而是被转移到了路由、负载均衡和通信上。训练时若少数专家被过度偏爱，会导致过载和不稳定；推理时则会表现为 batching 困难、尾延迟上升和专家并

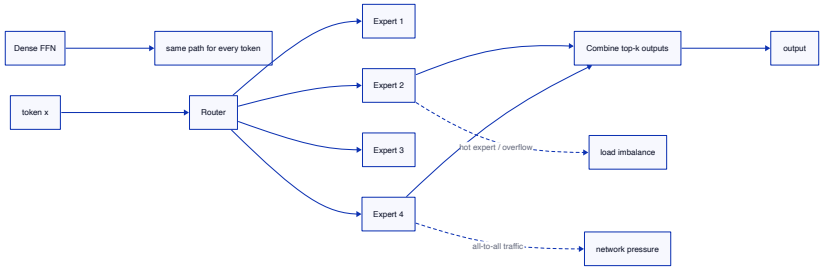

Figure 2

行通信变重。MoE 不是“便宜的大模型”，而是“把一部分密集计算成本换成路由复杂度的大模型”。

4 Encoder-Only 模型

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 什么是 encoder-only 模型？
> 2. 为什么 BERT 一类模型在分类、检索和 embedding 上依然很强？
> 3. 什么时候工程师应该优先用它，而不是聊天式 LLM？

encoder-only 模型的核心优势，是把输入压成高质量语义表示。它不追求像聊天模型那样持续续写，而是追求“读懂输入后，给出稳定的内部表征”。

4.1 BERT

BERT 是这一类架构的代表。它采用双向自注意力，让每个 token 在编码时同时读取左右文，再通过 MLM 目标恢复被遮盖的 token。这个目标天然更偏“理解”和“压缩语义”，而不是“从左到右持续生成文本”。

从工程角度看，BERT 风格模型长期有三类价值。

第一，它非常适合检索和排序。无论是生成 embedding、做双塔召回，还是做 cross-encoder reranking，encoder-only 都比大型 decoder-only 更高吞吐、更便宜，也更稳定。

第二，它适合分类和判别任务。内容审核、意图识别、实体分类、工单路由，这些问题并不需要模型写长答案，只需要模型稳定地区分语义类别。把生成式大模型硬塞进这种任务，往往只会增加延迟和成本。

第三，它适合作为系统的语义基础设施。很多 AI 产品真正消耗流量的不是“生成一段很长的回答”，而是持续地向量化文档、对话片段、代码块和查询。这里，BERT 一类模型常常是更现实

的底座。

它的局限也同样明确。BERT 不是原生生成器，不适合开放式续写、对话或工具调用。你当然可以在它之上再接解码器，但那已经不是“直接拿 BERT 来做聊天”。

工程选择建议：

- 当任务输出是标签、分数、排序或向量时，优先考虑 encoder-only；
- 当系统吞吐和单位成本比“像人说话”更重要时，优先考虑 encoder-only；
- 当任务要求开放式生成、多轮对话或程序式输出时，不要勉强使用 encoder-only。

5 Decoder-Only 模型

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 为什么 decoder-only 成了现代通用 LLM 的主流？
> 2. GPT、PaLM、LLaMA、Qwen、Mistral 这些家族，底层到底共享什么？
> 3. decoder-only 的统一性，换来的主要系统代价是什么？

decoder-only 的核心吸引力，是它把训练、对齐和推理全部统一成一个动作：给定前缀，继续生成下一个 **token**。这使它非常适合作为“通用语言接口”。

5.1 为什么它主导了现代 LLM

decoder-only 架构的胜利，并不来自某个单独技巧，而来自整个工程路径的统一。

- 预训练时，它学的是 next-token prediction；
- 指令微调时，它仍然是在给定上下文下继续生成；
- 工具调用时，本质上也是在生成结构化调用文本；
- 对话系统里，用户消息、工具结果、系统指令都能被拼成同一上下文前缀。

对于大规模训练和产品化系统来说，这种统一非常宝贵。你不需要为不同任务维护不同输入输出形状，也不需要在推理端重新发明另一套接口。

5.2 GPT 家族：统一生成接口的胜利

GPT 路线可以粗略看成一条持续强化“通用生成接口”的演化链：**GPT → GPT-2 → GPT-3 → GPT-3.5 / InstructGPT → GPT-4**。

- **GPT-2** 让业界看到，大规模自回归预训练已经能产生明显的零样本和少样本能力；
- **GPT-3** 把规模推高后，prompt-based few-shot learning 变成一种现实能力；
- **GPT-3.5 / InstructGPT** 说明“会续写”不够，还必须通过指令微调和 RLHF 让模型更像可用助手；
- **GPT-4** 的公开形象更强调系统级能力和多模态能力，但其内部架构细节并未完全公开，这本身就是一个工程提醒：不要把闭源模型的传闻当成可验证规格。

GPT 家族最重要的工程启示，不是“参数越来越大”，而是：一旦接口统一为自回归生成，预训练、对齐、工具使用、系统集成都可以沿同一骨架持续演化。

5.3 PaLM：大规模 decoder 的系统化配方

PaLM 的重要性在于，它展示了大规模 decoder-only 模型如何系统性地向更高效率演化。报告中一个很有代表性的设计，是把注意力和 FFN 作为并行 **Transformer** 子层处理，而不是完全串行展开。直觉上，这是在减少串行路径和部分通信开销。

PaLM 还同时采用了几项后来非常常见的设计：**SwiGLU**、**MQA**、**RoPE**、共享输入/输出嵌入。这些并不是为了“炫技”，而是分别在 FFN 表达效率、KV 缓存控制、位置建模和参数冗余上给出更现实的工程折中。

PaLM 的启示是：一旦模型走向大规模，真正重要的不再只是“Transformer 还是不是 Transformer”，而是每个 block 的数据流、注意力头配置、词表设计和内核友好性。

5.4 LLaMA：现代开源 decoder 配方的母体

LLaMA 的价值更像一个生态母体，而不只是某个单独 checkpoint。LLaMA 1、LLaMA 2、LLaMA 3 这条路线之所以重要，是因为它把一组非常实用的现代 decoder 配方推成了开源生态的共同语言：**RoPE**、**RMSNorm**、**SwiGLU**，以及后续更常见的 **GQA**。

这组配方并不戏剧化，但它带来了非常稳定的现实收益：

- 训练更稳；
- 推理成本更容易控制；
- 实现清晰，便于复用；
- 社区生态容易围绕它发展。

因此，大量后来的开源模型虽然名字不同，底层却都带着明显的“LLaMA-like”气质。工程师在读模型卡时，首先应该看的是它是不是沿用了这条现代 decoder 配方，而不是先看营销名称。

5.5 Qwen：同骨架下，数据与后训练如何拉开差距

Qwen 家族很好地说明了一个现实：共享骨架，不等于共享能力。从架构上看，Qwen 也属于现代 decoder-only 路线；以 **Qwen2** 为例，公开材料强调它在不同尺寸上采用 **GQA**，以改善推理效率和 KV 缓存占用；在上下文上，家族预训练通常以 **32K** 为基础，一些 instruction 版本再扩展到 **128K**。

Qwen2.5 则延续稠密 decoder 路线，并继续强化更长上下文、多语言、代码和指令能力。这里真正值得工程师记住的是：哪怕两个模型都写着“128K、decoder-only、GQA”，它们在中文、代码、工具使用和长文档 QA 上仍可能表现出完全不同的任务能力。差异往往来自数据构成、tokenizer、后训练和系统集成，而不仅是 block 公式。

5.6 Mistral：效率优先的 decoder 设计

Mistral 家族的工程特征很鲜明：它沿着 decoder-only 主线前进，但主动对推理与长序列成本做优化。代表性选择包括 **GQA**，以及部分模型采用的滑动窗口 / 局部注意力。这类做法的

收益很直接：降低部分计算与缓存压力，改善长序列吞吐；代价也同样直接：远距离 token 不再总能直接全局交互。

这就是为什么 Mistral 常被视为“效率优先”路线。它并不是从根本上放弃通用生成，而是反过来问：在保持通用 **decoder** 形态不变的前提下，哪里最值得做成本手术？

在这一家族里，**Mixtral** 是非常重要的延伸，它会在后面的 MoE 一节中详细展开。这里先记住一个结论：Mistral 代表稠密 decoder 的效率优化，Mixtral 代表在同一设计哲学下把稀疏专家引入系统。

5.7 Decoder-only 为什么仍然不是“默认万能答案”

尽管 GPT、PaLM、LLaMA、Qwen、Mistral 这些家族都围绕 decoder-only 收敛，但它的代价也极其稳定：

- 推理是串行生成，端到端延迟天然受限；
- 长上下文时 prefill 和 KV cache 成本迅速上升；
- 开放式生成更难做到严格格式控制；
- 在线部署时，显存和缓存管理经常先于参数量成为瓶颈。

因此，decoder-only 是今天最通用的主干，但不是每个任务的最优答案。它赢在统一性，不是赢在对所有任务都天然最省。

6 Encoder-Decoder 与 Hybrid 模型

> **!** Important
>
> 本节先回答几个关键问题：
> 1. T5 为什么采用 encoder-decoder，而不是像 GPT 一样只用 decoder？
> 2. 什么任务更适合把“理解输入”和“生成输出”拆开？
> 3. GLM 4.5 / 4.6 这类 PrefixLM 路线，为什么值得单独理解？

6.1 T5：当任务天然是“把一个输入变成另一个输出”

T5 的核心直觉是：当任务本质上是一个条件映射时，把输入和输出拆成两条不同计算路径通常更自然。encoder 先完整读取输入，得到稳定表示；decoder 再通过交叉注意力读取这些表示并生成输出。

这种设计特别适合翻译、摘要、改写、问答和结构化生成。因为输入和输出角色明确，decoder 不必在同一种上下文里同时兼顾“理解输入”和“继续自回归”两种不同工作。

T5 的另一个重要贡献，是把很多任务统一写成 **text-to-text** 接口。无论原始任务是分类、抽取还是摘要，最后都写成“输入字符串 $\rightarrow$ 输出字符串”。这让监督形式非常一致，也降低了任务定义碎片化。

它的代价也很清楚。相比同参数量的 decoder-only，encoder-decoder 通常需要额外跑完整个 encoder，并在生成阶段执行交叉注意力。因此对于开放式聊天这类持续生成任务，它往往不如 decoder-only 那么统一和轻便。

什么时候优先选 **T5**？

- 当任务是明确的输入到输出映射；
- 当输出结构较稳定，不需要开放式长对话；
- 当你希望条件生成更强，而不是希望模型扮演通用助手。

6.2 PrefixLM / GLM：介于理解与生成之间的折中

GLM 路线并不是传统的 encoder-decoder，但它回答了一个非常类似的问题：能不能让模型在生成之前，更充分地读取给定前缀？

在 PrefixLM / GLM 中，前缀部分允许更灵活的上下文读取，而生成部分仍维持因果一致性。因此它既不像纯 encoder-only 那样只做表示，也不像纯 decoder-only 那样从头到尾都严格左到右。它更像是在两者之间开出一条工程折中路线。

这类设计特别适合 infilling、blank filling 和带复杂上下文的条件续写。对工程师来说，理解 **GLM 4.5 / 4.6** 这类命名的关键，不是把它们看成“又一个模型名”，而是看懂它们背后的掩码哲学：在同一骨架里，让“读上下文”和“做生成”承担不同规则。

它的代价也很直接：掩码和训练逻辑更复杂，推理栈更难做成像纯 decoder-only 那样统一。因此，PrefixLM 是一条很有价值的中间路线，但它不会像 decoder-only 那样天然成为通用产品默认接口。

7 Dense 模型 vs Mixture-of-Experts（MoE）

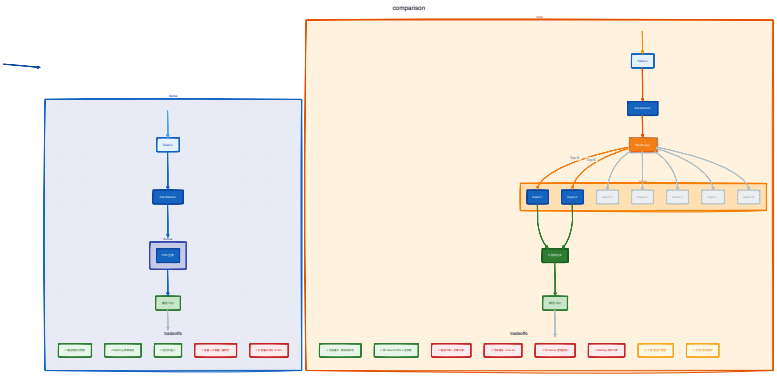

Figure 3: Dense vs MoE 计算路径：同样的容量目标，不同的代价分布

> **!** Important
>
> 本节先回答几个关键问题：
> 1. Dense 模型和 MoE 最大的工程差别是什么？

2. 为什么 Mixtral、DeepSeek-V2、DeepSeek-V3 会表现出“总参数很大，但每 token 激活没那么大”的特征？
3. 为什么 MoE 经常在离线评测和线上部署之间产生明显张力？
4. 为什么服务 MoE 时，平均吞吐和 P99 常常像两个不同系统？

Dense Transformer 的优点，是路径简单且稳定：每个 token 都通过完整网络，计算行为更容易预测，batching 也更容易做。代价则是容量和每 token FLOPs 强绑定，想扩大模型容量，通常就得线性增加计算。

MoE 试图把这件事拆开。总参数可以做得很大，但每个 token 只走少数专家，于是总容量和每 **token** 激活成本不再完全绑定。这是 MoE 最重要的工程价值。

Mixtral：**top-2** 路由为何成了一个现实折中

Mixtral 可以看作当前主流 MoE 设计的代表案例之一。像 8x7B top-2 这样的配置，总参数量很大，但每个 token 实际只激活两个专家，因此单 token 激活参数规模远小于总参数规模。

这带来两个直接收益：

- 在维持高总容量的同时，控制单次前向成本；
- 在某些预算下，让 MoE 比同等总参数的 dense 模型更现实。

但代价同样明确。路由本身要计算，专家之间要做更复杂的分配和通信；当不同 token 选择不同专家时，batching 和 tail latency 会更难控制。于是，Mixtral 一类模型在线上很容易出现“平均延迟还行，但 P99 很难看”的现象。

若把一批长度不同、内容不同的请求合并进同一 batch，top-2 路由的负载会表现出很强的输入依赖。平均激活参数量只说明单位 token 的理想计算量，并不说明服务调度的理想性。对实时系统来说，路由方差本身就是成本。

DeepSeek-V2 / DeepSeek-V3：**MoE** 不只是专家，还包括缓存与训练目标的再设计

DeepSeek-V2 和 **DeepSeek-V3** 值得单独理解，因为它们把 MoE 与别的系统优化绑在一起了。公开资料里，V2 被描述为 **236B** 总参数 / **21B** 每 **token** 激活，支持 **128K** context，并引入 **MLA**（**Multi-head Latent Attention**）与 **DeepSeekMoE**；V3 则进一步扩到 **671B** 总参数 / **37B** 每 **token** 激活、**128K** context，并加入 **MTP**（**Multi-Token Prediction**）以及无辅助损失负载均衡策略。

这里真正重要的不是记住数字，而是看懂它背后的工程意图：

- **MoE** 解决容量问题；
- **MLA** 继续压缩 KV 缓存；
- **MTP** 让训练监督更密，也为更激进的推理加速策略创造接口；
- 更复杂的负载均衡设计，说明 MoE 的瓶颈并不只在“有没有专家”，而在“专家会不会被均匀而稳定地使用”。

换句话说，DeepSeek V2 / V3 不是把几种新词堆在一起，而是在回答同一个问题：如何在不让单 **token** 成本失控的前提下，把容量、上下文和推理效率同时往上推。

DeepSeek-R1-Zero / DeepSeek-R1：架构不是全部，后训练也会重塑推理行为

DeepSeek-R1-Zero 和 **DeepSeek-R1** 的价值，更多来自后训练而不是骨架本身。公开材料说明，R1-Zero 展示了在没有传统 SFT 冷启动的情况下，仅靠强化学习也能诱导出较强的推理行为；而 R1 则在加入冷启动数据后改善了可读性与产品友好度，并建立在 **DeepSeek-V3-Base** 之上。

这给工程师一个非常重要的提醒：架构会决定成本结构和能力上限，但推理风格、可用性和对齐质量，往往还要由后训练阶段共同塑造。因此在看一个“reasoning model”时，不能只盯着 backbone 名字。

一个 **MoE** 事故模板：路由崩塌如何发生

一个典型的线上故障大致长这样：

- 症状：一两个专家持续变热，all-to-all 通信上升，P99 恶化，但平均吞吐和离线 benchmark 仍然看起来不错；
- 根因：真实流量分布与训练分布不同，router 在某类输入上高度偏置，导致专家利用率极不均衡；
- 缓解：更强的负载均衡约束、更保守的容量因子、路由温度调节、专家并行拓扑优化，以及对“热专家”做单独监控。

这类故障说明，MoE 容量不是白送的。它把“更大模型”的问题，从矩阵乘法变成了矩阵乘法加调度问题。

一个非常实用的判断

如果你的第一诉求是：

- 路径稳定、易于部署、延迟更可预测，优先 dense；
- 容量尽可能大，且愿意为路由与系统复杂度买单，可以考虑 MoE。

MoE 不是“更高级的 Transformer”，它只是另一种容量-成本交换方式。工程上真正该问的问题是：你的系统更怕算不动，还是更怕路由不稳。

8 Long-Context 架构

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 为什么“支持长上下文”同时是建模问题和系统问题？
> 2. 稀疏注意力、局部窗口、缓存压缩和位置外推各自在解决什么？
> 3. 为什么很多模型在配置上支持 128K，但在语义上并没有同等可靠的 128K？
> 4. 长上下文的真正服务预算应该如何算，而不是只看 model card？

长上下文是过去两年最容易被营销、也最容易被误解的架构主题之一。用户看到的是一个更大的窗口数字；工程师面对的则是三类同时出现的问题：

- 注意力 prefill 变重；
- KV 缓存快速膨胀；
- 位置泛化和检索质量开始退化。

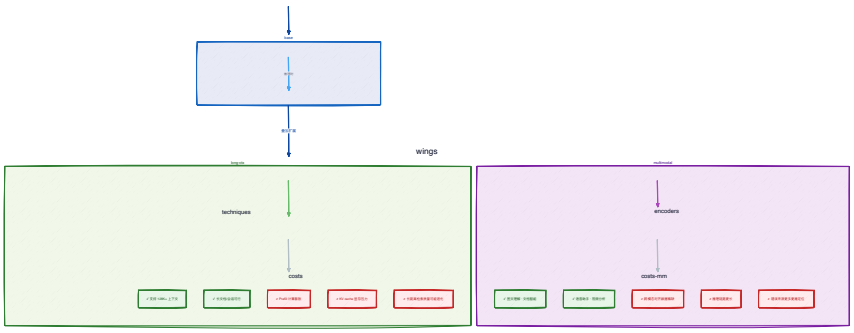

Figure 4: 长上下文与多模态：叠加到基础架构上的扩展层

也就是说，长上下文不是单一 feature，而是一组成本和能力的联合问题。

请求级成本如何增长

一个长上下文请求的端到端时延可以粗略写成：

$$T_{\text{req}} \approx T_{\text{prefill}}(n_{\text{in}}) + \sum_{t=1}^{n_{\text{out}}} T_{\text{decode}}(n_{\text{in}} + t)$$

其中：

- n_{in} 是输入长度；
- n_{out} 是输出长度；
- T_{prefill} 主要受注意力的二次复杂度影响；
- T_{decode} 则更常受 KV 读取和内存带宽影响。

这意味着，把上下文从 32K 拉到 128K，不只是 prefill 更重，后续每个输出 **token** 的 **decode** 也在更长历史上进行。

一个数值例子：为什么 **128K** 不是“只是 **4** 倍于 **32K**”

继续使用上一节的配置：

- $L = 80$；
- $G = 8$；
- $d_k = 128$；
- bf16，因此 $b = 2$；
- 单请求先取 $B = 1$。

前面已经算过，32K 时总 KV 约 10 GiB，128K 时约 40 GiB。看起来只是线性增长四倍，但在服务里这四倍往往会触发连锁反应：

1. 单请求剩余显存更少，导致并发数下降；

2. 批处理更难打齐，因为长短请求混在一起会放大尾部；
3. **prefill** 时间上升更快，让用户感知到明显首 token 延迟；
4. 缓存和权重共同争用带宽，decode 吞吐也会随之下降。

因此，很多团队第一次把 32K 升到 128K 时，会出现一个典型现象：离线单请求测试“能跑”，上线后 p95/p99 却突然变差。

路线一：继续用稠密注意力，但把实现做得更“可服务”

这是最保守也最常见的路径。模型仍保留全局稠密注意力，但在内核、缓存分页和位置外推上做优化，例如 FlashAttention、PagedAttention、一系列 RoPE scaling 技巧。它的优势是骨架不变，生态友好；劣势是二次复杂度本身没有消失，只是被更高效地实现。

这条路常见于许多现代 decoder-only 家族，包括 **LLaMA** 路线上的大量模型，以及 **Qwen2 / Qwen2.5** 这类把预训练上下文推到更长区间、再辅以更长 instruction 版本的做法。

路线二：改变注意力连接图

另一类方法会改变 token 之间“谁能直接看谁”。例如 **Mistral** 一些模型使用的滑动窗口 / 局部注意力，本质上是让 token 重点关注局部窗口，而不是永远看完整个历史。收益是更低的部分计算和缓存开销；代价是长距离依赖不再总有直接路径。

这类设计适合那些局部模式占主导的任务，但如果产品高度依赖跨很长文档的精确引用、远距检索和全局一致性，局部窗口会把问题暴露出来。

路线三：更激进地压缩缓存表示

DeepSeek-V2 / V3 中的 **MLA** 是很好的代表。与 GQA 通过减少 KV 组数来压缓存不同，MLA 更进一步地把 KV 表示压到潜在空间里，再在需要时恢复。收益是长上下文服务的缓存开销继续下降；代价是实现、训练和推理内核都更复杂。

这说明长上下文架构不只是在“注意力图”上动手，也可以在“缓存表示”上动手。很多时候，系统真正的瓶颈不是参数量，而是缓存如何活得下去。

路线四：原生长上下文训练，而不是只做外推

有些模型会通过更长序列训练，真正让模型见过更长的距离分布，而不是仅仅依赖 RoPE 缩放去“硬拉”窗口。工程上，这条路成本最高，但通常也最可靠。它的真正价值不是把窗口数字变大，而是让模型在长距离检索、引用和推理时更少退化。

长上下文里，最常先出问题的不是参数量，而是显存与带宽。prefill 吞吐、KV cache、并发调度、prefix sharing、batch packing，这些服务系统细节经常比“模型是不是 70B”更决定产品体验。也因此，长上下文模型是否可用，必须同时回答两个问题：

1. 数学上能不能输入这么长；
2. 生产里能不能在目标延迟下稳定服务这么长。

如果你的任务本质上是“从大语料里找少量相关事实”，先做 **retrieval**，再考虑纯长上下文。把检索问题硬变成长窗口问题，通常只会把成本和错误一起放大。

9 多模态架构

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 多模态模型并不是“多接一个输入”，那它到底在改什么？
> 2. 图像与音频编码器通常如何与语言模型整合？
> 3. 为什么跨模态对齐错误经常比单模态语言错误更难定位？
> 4. 为什么一个 OCR 看起来正确的系统，仍然可能在图文推理上失败？

多模态架构解决的是一个根本问题：语言模型天然只吃 **token**，而现实世界里的输入并不是天然就是 **token**。图片、音频、视频、版面、表格，都必须先经过编码，然后被稳定地接入语言主干。

把它写成最常见的数学形式，大致是：

$$z_v = E_v(x_v), \qquad h_v = W_p z_v$$

其中 E_v 是视觉或音频编码器，x_v 是原始模态输入，W_p 是投影器或适配器，h_v 则是语言主干可消费的“伪 token”表示。

随后系统一般走两条路：

- 拼接式融合：把 h_v 直接拼到文本 token 前后，再由 decoder-only 主干统一处理；
- 交叉注意力融合：让语言 token 在生成时通过 cross-attention 读取视觉表示：

$$\mathrm{CrossAttn}(Q_t, K_v, V_v) = \mathrm{softmax}\left(\frac{Q_t K_v^\top}{\sqrt{d}}\right) V_v$$

从架构角度看，多模态真正改变的不是“输入框里多了图片”，而是：主干是否有稳定的跨模态接口，训练时是否学会了把这些接口对齐到语言任务。

1. 模态编码器 ＋ 投影器 ＋ 语言模型

这是最常见的系统形态。视觉编码器或音频编码器先把原始信号变成连续表示，再通过线性投影或小型适配器，把这些表示映射到语言模型可消费的隐藏空间。这样做的好处是可以复用成熟的语言主干；问题在于跨模态对齐的质量，往往完全取决于中间这层桥接是否足够好。

这条路线在文档理解里尤其常见：页面图像先经过视觉编码，版面或 OCR 特征再被投到语言空间，最后由 LLM 生成问答、抽取结果或解释文本。

2. 交叉注意力的多模态生成器

另一条路线会显式引入 cross-attention，让语言解码器在生成时读取来自图像或音频编码器的表示。这在结构上更接近把“视觉 encoder ＋ 文本 decoder”拼在一起，适合条件生成任务，但推理路径更长，训练也更依赖高质量对齐数据。

这类结构在图像描述、视觉问答、视频摘要和语音条件生成里都很自然，因为视觉 / 音频信息被明确当作“条件”。

3. 更统一的 token 化或共享主干

更激进的方法会尝试把不同模态尽量统一成同一种 token 或近似同一种序列接口，让单一 Transformer 主干处理多类输入。这条路在理论上很优雅，但工程门槛也最高，因为不同模态的统计结构和时间尺度差异很大。

它的潜在优点，是统一主干、统一训练目标、统一推理接口；它的代价，则是统一表示未必等于统一可用性。很多模态在物理上并不天然适合被当作语言 token。

> **Note**
>
> 案例分析
> 很多团队第一次做多模态文档系统时，会出现这样一种困惑：
> - OCR 文本看起来几乎全对；
> - 模型也能复述页面大意；
> - 但一旦问题变成“表格中第三列和页脚条件一起决定的金额是多少”，错误率就明显上升。
>
> 这类故障的根因，通常不在“模型不会读字”，而在视觉特征与语言指令没有被稳定对齐。可能的问题包括：
> - 投影器把局部视觉关系压得过于平坦，丢失了空间结构；
> - 模型看到的是 OCR 串接文本，而不是版面关系；
> - 训练数据主要奖励“看懂局部图像”，没有足够奖励“跨区域联合推理”。
>
> 也因此，文档理解系统经常需要额外的 2D 位置编码、区域 token、表格结构化抽取，甚至专门的版面 encoder。多模态系统的失败，常常不是“没看见”，而是“看见了却没和语言任务对齐”。
> 从本章角度，真正重要的不是背出某个视觉语言模型名字，而是理解：多模态架构的代价来自跨模态接口，而不是来自语言主干本身。
> 值得注意的是，**GPT-4** 的公开产品形态已经表现出明显的多模态系统能力，但其具体内部架构并未完全公开。这再次提醒我们：在闭源系统上，可以分析公开行为和系统接口，但不应该捏造不可验证的内部实现。

10 如何评估模型架构

> **Important**
>
> 本节先回答几个关键问题：
> 1. 工程师应该如何比较不同架构，而不是只盯着排行榜？
> 2. benchmark、任务评测和效率指标分别在解决什么问题？
> 3. 为什么同一个高分模型仍可能不适合你的产品？

评估架构，至少要同时看三层：

1. 能力基准：模型会不会做；
2. 效率指标：模型做这件事要花多少时间、显存和钱；
3. 任务贴合度：它在你的真实场景中会不会以正确方式失败。

前两层告诉你“理论上可不可以”，第三层告诉你“产品上值不值得”。

10.1 能力基准

这一层看的是模型在公开 benchmark 上能做什么。对通用 LLM，一个实用的最小集合通常应覆盖：

- 通识与推理：MMLU / MMLU-Pro、GPQA、BBH；
- 数学：GSM8K、MATH；
- 代码：HumanEval、MBPP、SWE-bench；
- 指令跟随 / 对话：MT-Bench、AlpacaEval、LMSYS Arena；
- 真实性 / 安全：TruthfulQA，配合 red teaming；
- 多语言：MGSM、TyDiQA、XQuAD；
- 长上下文：LongBench、Needle-in-a-haystack；
- 多模态：MMMU、MathVista、MMBench。

这些 benchmark 的价值，在于提供公共坐标系；它们的局限，也同样明显：分布静态、容易被适应、未必贴合你的任务。

10.1.1 效率指标

同样准确的两个模型，可能在生产上完全不是一个量级。架构评估必须显式纳入：

- prefill 吞吐；
- decode 吞吐；
- P50 / P95 / P99 延迟；
- 每请求显存占用；
- KV cache 增长曲线；
- 批处理效率与并发上限；
- MoE 专家通信与尾延迟；
- 训练时的 tokens/sec 与集群扩展效率。

这类指标不是“部署之后再看”的附属信息，而是架构优劣的一部分。一个在离线基准上更强、但 P99 高得无法上线的模型，工程上并不更优。

10.1.2 任务贴合度

最终决定是否选择某个架构的，往往不是总榜，而是：

- 你的任务更像检索、翻译、对话，还是 Agent？
- 你的用户输入是短问答，还是长文档与长会话？
- 你更怕 hallucination、延迟、成本，还是推理失败？
- 你是否需要图像、音频、表格、版面理解？

这就是为什么 benchmark 永远只是起点。真正的选型应该先从产品失败模式出发，再看哪个架构最适合承受这些失败模式。

10.1.3 评测方法

评估方法可以粗略分成四类，它们回答的是不同层面的“好不好”。

1. 离线自动评估（**offline evaluation**）
   优点是快、便宜、可复现，适合知识、数学、代码等有明确答案或明确判分规则的任务。缺点是容易被 prompt 格式、数据污染和 benchmark overfitting 影响。

2. 模型评判（**LLM-as-a-judge**）
   适合规模化比较写作质量、指令跟随、回答完整性等不易自动打分的任务。问题在于它常有位置偏差、冗长度偏差、风格偏差，以及 judge 模型版本漂移。缓解方式包括：随机化候选顺序、盲化模型身份、固定 judge 版本、用小规模人工标注集做校准。

3. 人类偏好测试
   当你真正关心的是用户体验、语气、帮助性、安全边界和产品满意度时，人类测试通常仍是最可信的。代价是昂贵、缓慢，而且需要认真设计采样与标注规范，否则人类评估本身也会很噪声。

4. 系统级 / **agent** 级评估
   这类评估看的是端到端任务成功率，例如工具调用是否正确、多轮任务是否完成、代码库修改是否可合并。它们最接近真实价值，但也最难标准化，基础设施成本最高。

一个常见误区是把 LLM judge 当作“便宜的人类替代”。更准确的理解是：**LLM judge** 是一种高吞吐近似测量工具，而不是最终真理。当任务涉及风格偏好、品牌语气、医疗或法律等高风险决策时，人类评估仍然不可替代。

10.2 常见评测陷阱

很多“模型 A 比模型 B 强”的结论，最后不是败给模型本身，而是败给评测设置。最常见的陷阱包括：

- **Prompting** 不一致：0-shot、few-shot、CoT、system prompt、模板格式稍有变化，分数都可能波动；
- **Decoding** 不一致：temperature、top-p、是否固定 seed、是否允许多次采样，都会改变结果；
- 污染（**contamination**）：训练集与测试集重合、题目被记住、公开 benchmark 被反复优化；
- 评分方法不一致：exact match、F1、pass@k、LLM judge、人类偏好，各自衡量不同维度，不能直接混用。

“MMLU 很高，但真实客户体验一般”的根源，通常是分布不匹配。MMLU 更像整洁的选择题知识测验，而客户问题往往是多轮、含糊、上下文依赖、需要工具或检索、还带着风格和安全约束。模型能在一个静态、格式规范的 benchmark 上表现好，不代表它已经适应真实产品中的噪声与工作流。

10.3 工具、排行榜与第一手资料

一个成熟的评估工作流通常同时依赖三类资源：

- 评测套件 / **harness**：OpenCompass、lm-evaluation-harness、HELM；
- 排行榜 / 聚合站点：Hugging Face Open LLM Leaderboard、LMSYS Chatbot Arena、Papers with Code；
- 第一手规格资料：技术报告、model card、配置文件和推理仓库。

评测套件解决的是执行一致性；排行榜解决的是公共对比坐标；第一手规格资料解决的是事实一致性。三者缺一不可。

10.4 推荐的最小基准集合

一个可操作的经验是：用小而多样的集合，而不是用一个巨大的总榜。对通用 LLM 来说，最低限度也应该覆盖通识 / 推理、数学、代码、指令跟随 / 对话质量、真实性 / 安全、多语言，以及长上下文；如果模型带视觉能力，再补充多模态评测。

不要依赖单一数字的原因很简单：不同 benchmark 的输入形式、判分方法和能力指向完全不同。把它们粗暴求和，往往只会把真实差异抹平。

这一节里最常出现的两个公式是：

- 困惑度（Perplexity）：

$$\mathrm{PPL} = \exp\left(\frac{1}{n}\sum_{t=1}^{n} -\log p(x_t \mid x_{<t})\right)$$

  困惑度衡量模型在给定分布上“下一个 token 预测得有多不确定”。它适合理解语言建模质量，却不等价于指令跟随、对齐或工具使用能力。

- 代码中的 pass@k：若从模型采样 n 个程序，其中 c 个通过测试，则一个常用的无偏估计写成：

$$\mathrm{pass@}k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$$

  它比 pass@1 更接近“给模型多次机会”的真实使用方式，但也会把采样预算带来的收益混入模型能力本身。

下面这组 benchmark，可以作为一个实用起点，而不是最终答案。

通识与推理

- MMLU / MMLU-Pro
- GPQA（困难科学推理）
- BBH（Big-Bench Hard）

这一组 benchmark 的作用，是给模型的通识储备和通用推理能力画一个“基础轮廓”。其中，**MMLU** 更像覆盖广泛的知识选择题集合，适合做横向基线；**MMLU-Pro** 更强调更难、更不容易被简单模板技巧拿下的版本，因此分数通常不能直接和原始 MMLU 混看。

GPQA 补的是高难科学推理，能把“会做常规知识题”和“能在陌生复杂科学问题上稳住”区分开；**BBH** 则覆盖更杂的困难任务，帮助你观察模型在多种推理形态上的边界。

数学

- GSM8K（小学到初中水平数学）
- MATH（竞赛风格数学）

这两个基准覆盖的是不同难度层级。**GSM8K** 更适合观察模型在较短链条上的基础数学推理；**MATH** 则会更快暴露长推理链、符号操作和中间步骤稳定性的问题。只看其中一个，很容易误判模型的数学边界。

数学评测还有一个常被忽略的变量：自一致性或多数投票往往能显著提升最终正确率。这时你要分清楚：你评的是“单次输出质量”，还是“给定推理预算后的求解能力”。这两者在产品里对应完全不同的成本结构。

代码

- HumanEval
- MBPP
- SWE-bench

在代码领域，**pass@1** 测的是模型第一次输出就写对的概率；**pass@k** 测的是给模型 k 次采样机会后，至少有一个候选能通过测试的概率。前者更接近“单次自动补全”的体验，后者更接近“让模型多试几次再筛选”的工作流。

HumanEval 与 MBPP 更偏静态函数级任务，适合快速观察代码生成基本功；**SWE-bench** 更接近真实工程修复，需要在代码库上下文里理解问题、改动文件并通过测试。两者都重要，因为一个模型可能很会写短函数，却不擅长在真实项目里定位和修复错误。

指令跟随 / 对话质量

- MT-Bench
- AlpacaEval
- LMSYS Arena

这组 benchmark 解决的问题不是“会不会答题”，而是“答得像不像一个可用的助手”。**MT-Bench** 和 **AlpacaEval** 更适合结构化比较指令跟随和回答质量；**LMSYS Arena** 则通过人类盲测对战，把开放式对话质量转成相对排名。

Elo 排名之所以有价值，是因为对话质量很难被一个静态参考答案穷尽，成对比较往往比绝对分数更稳定。但 Elo 也不能替代产品指标。面向客户的聊天机器人，内部评估至少还应覆盖：准确性、安全性、风格一致性、拒答边界、延迟、成本以及工具调用成功率。

真实性 / 安全

- TruthfulQA
- 安全红队：先定义清晰的策略 / 威胁模型，再结合自动扫描与人工构造数据；示例工具包括 garak 和 NVIDIA 在 Nemotron-4 model card 中引用的 AEGIS

TruthfulQA 关注的不是一般事实问答，而是模型是否会迎合常见误解、迷信或错误先验。也就是说，它测的是一种很特殊、却在真实产品里很关键的失败方式：模型为了“像个答案”而复述大众错误。

安全评估则更不能只靠单一题库。你需要先明确威胁模型：担心的是越狱、仇恨内容、自伤建议、隐私泄露，还是工具滥用？然后再结合自动化扫描、人工红队、对抗 prompt、上下文注入和系统级压力测试。

多语言

- MGSM
- TyDiQA
- XQuAD

多语言评测的意义，在于检验模型能力是否真的跨语言迁移，而不是只在英文上成立。**MGSM** 适合看跨语言数学推理是否稳定；**TyDiQA** 更强调多语言问答与信息理解；**XQuAD** 则提供跨语言阅读理解对比。

为什么这组评测不能省？因为多语言能力不仅受模型规模影响，还受 tokenizer、语料比例、脚本类型和后训练语言覆盖影响。一个英文表现极强的模型，完全可能在中文长问答、阿拉伯语搜索问答或代码与自然语言混写场景下掉出另一条曲线。

长上下文

- LongBench
- Needle-in-a-haystack：使用受控的合成检索探针；一个常被引用的公开实现是 https://github.com/gkamradt/LLMTest_NeedleInAHaystack

长上下文评测至少要分成两类。**Needle-in-a-haystack** 测的是：把关键信息埋在很长文本里，模型能否在不同位置把它捞出来；它强调受控、可解释，适合检查“检索能力是否还在”。**LongBench** 则更像一组长输入任务集合，覆盖摘要、问答、理解等更接近真实使用的场景。

两者必须一起看，因为只会“捞针”不代表能在长文档任务里稳定工作，只会做长任务平均分也不代表定位能力可靠。长上下文最常见的误判，就是把“数学上能输入这么长”误当成“语义上还能用这么长”。

多模态

- MMMU
- MathVista
- MMBench

多模态评测的关键，是区分“看见了什么”和“理解了什么”。**MMMU** 更强调跨学科知识与多模态推理；**MathVista** 更强调图表、几何、视觉数学推理；**MMBench** 则提供较为综合的视觉语言能力对比。

如果只测 OCR 或图片描述，你只能知道模型能不能把图像转成文字，不能知道它能否围绕图像做推理、比较、计算和任务决策。真正的多模态系统，应该在视觉感知、文本理解、跨模态对齐和下游任务完成之间形成闭环；相应地，评测也必须覆盖这些层次。

> **!** Important
>
> 为什么 **benchmark** 不足以替代架构理解
> 因为 benchmark 多半回答的是“在一组题上表现怎样”，而架构理解回答的是“它为什么会这样表现、上线后会在哪里出问题”。
> 一个模型在 MMLU、GPQA 或 Arena 上高分，并不自动意味着：
> - 它适合长会话客服；
> - 它的 KV cache 能支撑你的并发；
> - 它在你的中文工单、代码仓库或图文文档上仍然稳；
> - 它的 P99 延迟在你的服务预算之内。
>
> 工程师真正需要的，不是一个“更强模型”的抽象结论，而是一种把架构特征映射到系统约束的能力。

11 架构究竟改变了什么

> **Important**
>
> 本节先回答三个问题：
>
> 1. 为什么架构不只是理论差异，而是系统行为差异？
> 2. 一个架构选择，会通过哪些路径影响训练和推理？
> 3. 为什么“能力强”与“可部署”经常是两套不同的约束？

从系统视角看，模型架构主要改变五件事。

第一，信息流方向。双向编码器允许每个 token 同时读取左右文，更适合理解和压缩语义；因果解码器只能看左边，更适合持续生成；encoder-decoder 则把“读输入”和“写输出”拆成两条不同计算路径。

第二，参数如何被激活。Dense 模型让每个 token 都通过完整网络；MoE 只激活少数专家。前者执行路径稳定，后者更像用路由复杂度交换模型容量。

第三，状态如何随序列增长。长上下文不是“能多塞一些 token”这么简单，它意味着更重的 prefill、更大的 KV 缓存、更高的显存带宽压力，以及更难控制的并发行为。

第四，外部模态如何接入。多模态模型并不是简单地“多接一个图片输入框”，而是要把图像、音频、视频编码成能被语言主干消费的表示，并在训练和推理时维持跨模态对齐。

第五，系统瓶颈会出现在哪里。有些架构更吃训练算力，有些更吃推理显存，有些更依赖 all-to-all 通信，有些更依赖位置外推和缓存管理。工程师选架构，本质上是在选择自己的主瓶颈。

架构选择	改变的核心对象	对训练的影响	对推理/部署的影响	更适合的系统
Encoder-only	表示学习与双向上下文	更适合判别与语义表示任务	一次编码即可，高吞吐	检索、分类、排序、embedding
Decoder-only	自回归生成能力	训练目标与推理接口高度统一	解码串行，KV 缓存随长度增长	对话、代码、Agent、通用生成
Encoder-decoder	条件生成映射	输入输出角色清晰，监督形式自然	需同时维护 encoder/decoder 计算	翻译、摘要、改写、结构化输出
PrefixLM / 混合掩码	理解与生成的折中	掩码设计更灵活，也更复杂	推理与实现逻辑更难统一	Infilling、条件续写、复杂前缀读取
MoE	激活参数量	容量更大，但路由更难训	延迟更不稳定，通信更复杂	大规模高容量模型
长上下文机制	序列长度与远距依赖	长序列训练与位置泛化更难	Prefill 与 KV 缓存压力上升	长文档、长会话、代码库级任务

架构选择	改变的核心对象	对训练的影响	对推理/部署的影响	更适合的系统
多模态接入	输入类型与对齐方式	训练更依赖模态对齐数据	推理链路更长，错误来源更多	视觉理解、语音助手、文档智能

12 工程案例

> **!** Important
>
> 下面不是单一公司的事故复盘，而是行业里反复出现的复合型故障模式。每个案例都回答三个问题：症状是什么，根因是什么，工程教训是什么。

12.1 把聊天模型拿去做 reranker，质量没上去，成本先爆了

症状：团队用一个大聊天模型给搜索结果逐条打分，单请求延迟显著上升，GPU 费用被放大，排序结果却并没有比一个强 encoder-only reranker 更稳定。

根因：架构错位。排序问题本质上是表示与判别问题，需要稳定的相关性评分和高吞吐；decoder-only 模型却在做串行生成式推理，既慢，又更难得到一致分数。

工程教训：架构要跟输出形状走，而不是跟榜单热度走。向量、标签、分数优先考虑 encoder-only；开放式文本才优先考虑 decoder-only。

12.2 MoE 模型离线很强，线上尾延迟却失控

症状：离线 benchmark 非常漂亮，平均 decode 吞吐也不差，但线上一到高并发，P99 急剧上升，批处理效率下降，GPU 利用率并没有预期中那么高。

根因：MoE 路由让不同 token 走向不同专家，导致专家负载不均、通信模式复杂、尾部请求更难被整齐打包。平均值看起来没问题，但 tail latency 被路由复杂度放大了。

工程教训：MoE 选型不能只看“激活参数更省”，必须同步评估路由均衡、专家并行通信、批处理策略和 P99。对许多实时系统来说，可预测性本身就是一项能力。

12.3 128K 写在 model card 上，系统却在 128K 里死掉了

症状：团队把上下文窗口从 32K 提到 128K，希望减少检索和摘要管线。结果首 token 延迟翻倍，并发数下降，长文档 QA 还经常找不到埋在中间段落的关键信息。

根因：窗口增大带来了 prefill 和 KV cache 的系统成本，同时模型并没有在同等长度上学到足够稳定的检索与引用行为。于是“数学上能放下”并没有变成“语义上更会用”。

工程教训：长上下文是能力与成本同时上升的特性，绝不是免费的。上线前必须同时做长上下文质量评测和服务压测，不能只看最大窗口数字。

12.4 公开榜单前排模型，客服工单却解不掉

症状：模型在 MMLU、Arena 上排名很亮眼，但进入真实客服流程后，经常答非所问、不会引用内部规则、工具调用失败率高。

根因：公开 benchmark 与真实工单分布差异很大。前者是整洁题目或通用对话，后者是多轮、带上下文噪声、带系统约束、需要检索和工具调用的任务。架构虽然强，但产品链路没有围绕真实失败模式被验证。

工程教训：排行榜适合筛选候选，不适合直接做上线决策。真正的产品评估必须围绕任务、工具、上下文和风格约束来设计。

12.5 多模态 demo 很惊艳，文档理解却不可靠

症状：模型做图片问答 demo 很亮眼，但一进入发票、截图、表格、版面复杂文档，错误率明显升高，尤其在跨区域引用和图文联合推理上频繁失真。

根因：多模态系统的薄弱点不总在语言主干，而常常出在模态编码器、投影层和跨模态对齐上。模型“看见了某些东西”，不等于它能在结构化版面里稳定地把视觉信号和语言任务对齐。

工程教训：评估多模态架构时，不要只做 OCR 或简单图片描述测试。必须覆盖版面理解、跨区域引用、视觉推理和端到端任务完成。

13 本章小结

面对模型选型时，一个实用做法是先回答四个问题：

1. 输出形状是什么？是向量、分数、标签，还是开放式文本？
2. 主要瓶颈是什么？是训练算力、推理显存、端到端延迟，还是工程复杂度？
3. 上下文有多长？你要的是 4K 的稳定服务，还是 128K 的离线分析？
4. 是否需要外部模态？只是文本，还是图像、音频、表格、版面一起进入系统？

如果用一个更抽象的工程视角看，可以把架构选型理解为在五个维度上取最小坏处：

- 质量上限；
- 延迟与吞吐；
- 显存 / 带宽压力；
- 实现与运维复杂度；
- 安全与可调试性。

架构原型	代表模型 / 家族	主导优势	主导成本	何时优先选择	常见误用
Encoder-only	BERT	表示学习强，高吞吐，延迟低	不擅长开放式生成	检索、分类、reranker、embedding	拿去做长对话或开放式写作

架构原型	代表模型 / 家族	主导优势	主导成本	何时优先选择	常见误用
Decoder-only	GPT、PaLM、LLaMA、Qwen、Mistral	接口统一，生成能力强，最适合通用助手	解码串行，KV cache 随长度增长	对话、代码、Agent、自由生成	拿去替代所有判别任务
Encoder-decoder	T5	输入输出角色清晰，条件生成自然	推理链路更长，部署更复杂	翻译、摘要、改写、结构化抽取	当成通用聊天骨干
PrefixLM / Hybrid	GLM 4.5 / 4.6	在理解与生成之间做精细折中	掩码逻辑和推理栈更复杂	infilling、复杂条件续写	以为它天然能替代 encoder-decoder
Dense	稠密	路径稳定，可预测性高	容量和单 token 计算强绑定	实时系统、低尾延迟服务	用固定预算硬堆更大 dense 模型
MoE-free	LLaMA / Qwen / Mistral				
MoE	Mixtral、DeepSeek-V2、DeepSeek-V3	以较低激活成本扩总容量	路由、负载均衡、通信复杂	高容量推理、大模型压成本	只看平均延迟，不看 P99
Long-context 变体	长窗 Qwen2/2.5、LLaMA 路线、Mistral 窗口、DeepSeek MLA	能处理长文档和长会话	Prefill、KV cache、位置泛化一起变难	长文档 QA、代码库级分析	以为“窗口更大”就等于“质量更好”
Multimodal	GPT-4 类产品形态、各类 VLM / Speech-LM	输入类型更丰富	对齐、投影器、跨模态故障更多	图像问答、文档理解、语音助手	只测 OCR，不测推理与 grounding

一个很实用的经验法则是：

- 输出是向量或分数时，先想 **encoder-only**；
- 输出是文本并且接口要通用时，先想 **decoder-only**；
- 输入到输出映射高度明确时，先想 **encoder-decoder**；
- 容量是第一诉求、且能承受路由复杂度时，再想 **MoE**；
- 长上下文与多模态永远不是免费升级，它们会把系统瓶颈暴露得更早。

13.1 问题小结

1. 什么是模型架构？它和参数量大小有什么区别？
模型架构描述的是计算图如何组织：信息如何流动、参数如何激活、上下文如何读取、输出如何生成；参数量只是这张计算图上的规模，不说明路径设计。

2. 为什么 **Transformer** 成为现代语言模型的默认骨架？
因为注意力让模型能动态建模 token 之间的关系，扩展性强、并行训练友好，也更容易和长上下文、多模态、MoE 等后续设计组合。

3. encoder-only、**decoder-only**、**encoder-decoder** 与 **PrefixLM** 的本质区别是什么？
encoder-only 强调双向理解与表示；decoder-only 强调因果生成；encoder-decoder 把输入理解和输出生成拆成两条路径；PrefixLM 在同一骨架里对“读取前缀”和“生成后缀”施加不同掩码规则。

4. 为什么大多数通用 **LLM** 选择 **decoder-only**？
因为 next-token prediction 把预训练、指令微调、工具调用和推理接口统一成一个形式，工程扩展路径最干净。

5. 模型架构设计通常受哪些约束驱动？
训练可扩展性、推理延迟、显存、上下文长度、模态接入、对齐方式以及目标产品任务形状。

6. Dense 与 **MoE** 的核心权衡是什么？
Dense 路径稳定、易部署，但容量和计算绑定；MoE 能以较低单 token 激活成本扩展容量，但引入路由、负载均衡和通信复杂度。

7. 为什么有些模型支持更长上下文？
因为它们在位置建模、长序列训练、注意力图或缓存表示上做了专门设计；长上下文不是单靠一个配置项自动得到的。

8. 注意力结构如何影响计算成本？
全局稠密注意力带来 $O(n^2)$ prefill 成本；MHA/MQA/GQA 影响 KV cache；局部注意力和缓存压缩则改变长序列的服务成本结构。

9. 哪些架构选择通常有利于 **reasoning**？
更强的 backbone 能提供上限，但 reasoning 行为往往还依赖训练目标、长过程监督和后训练；架构与后训练必须一起看。

10. 为什么有些模型更容易扩展到多模态？
因为它们的主干和接口更容易接收外部编码器表示，或者原生支持 cross-attention / 统一 token 接口。

11. 为什么一个模型写着 **128K context**，仍可能在长文档 **QA** 上失败？
因为“能放下 128K token”只说明数学窗口大小，不说明模型已经学会在该长度上稳定检索、定位和引用事实；同时服务系统还可能被 prefill 与 KV cache 压垮。

12. 生产系统里如何选择模型架构？
先看任务形状和失败模式，再看成本与延迟约束，最后才看公开总榜。任务若以检索和分类为主，优先 encoder-only；若以开放式生成为主，优先 decoder-only；若是稳定条件映射，优先 encoder-decoder。

13. 模型尺寸、激活参数量、**KV cache** 和延迟是什么关系？
总参数决定加载和训练规模；激活参数量决定单 token 计算成本；KV cache 决定长上下文与并发时的显存压力；端到端延迟则由 prefill、decode 和调度共同决定。

14. 什么时候应在生产里优先选择 **encoder-only**，而不是 **decoder-only**？
当输出是分数、标签、向量或排序，而不是开放式文本；当吞吐、成本和稳定性比“像人说话”更重要时，应优先 encoder-only。

15. 为什么不同架构在不同任务中长期占优？
因为任务要求的信息流不同。检索要求表示稳定，翻译要求输入输出映射清晰，对话要求统一生成接口，多模态要求模态桥接稳定。

16. 服务 **MoE** 模型时，会出现哪些 **dense** 模型没有的新故障模式？
热点专家、容量溢出、all-to-all 通信放大、batching 失配、P99 急剧恶化，以及流量分布变化导致的路由崩塌。

17. 当前架构演化的主要趋势是什么？
一是现代 decoder 配方继续收敛；二是 MoE 与缓存压缩继续用于扩容量和控成本；三是长上下文从“配置扩窗”走向更系统的训练与服务协同；四是多模态接口成为主干设计的重要部分。

13.2 给 AI 工程师的实践清单

Figure 5: 模型选择矩阵：任务 × 架构 × 成本 — 速查页

任务形状	默认优先架构	为什么	重点风险
语义检索 / reranking / 分类	Encoder-only (BERT 类)	输出是向量、分数或标签；高吞吐更重要	误用成开放式生成器
通用助手 / 代码补全 / Agent	Decoder-only (GPT / LLaMA / Qwen / Mistral 类)	接口统一，最适合连续生成和工具调用	KV cache、串行 decode、长会话成本
翻译 / 摘要 / 改写 / 结构化抽取	Encoder-decoder (T5 类)	输入输出角色清晰，条件生成自然	推理链路更长
极大容量但预算受限	MoE (Mixtral / DeepSeek-V2 / V3 类)	总容量大，激活成本相对可控	路由、通信、P99、调度复杂
长文档问答 / 长会话	长上下文架构 + 检索	既需要窗口，也需要 retrieval	只扩窗口、不做服务预算
图文理解 / 文档智能 / 语音助手	多模态架构	必须稳定接入视觉或音频表征	对齐、投影器、grounding 失败

在真正的产品团队里，选型很少是一锤定音。更现实的做法通常是：

- 先用最便宜、最稳定的架构做 baseline；
- 再证明更复杂的架构在你的任务上确实有增益；
- 最后才为这份增益支付训练、服务和运维成本。

这也是为什么，架构理解不是“知道有哪些模型”，而是知道什么时候不该升级。

本章真正要建立的，不是“认识几个模型名”，而是一种更稳定的工程视角。

1. 架构决定系统约束。不要把模型当成抽象“能力黑盒”；先问它的注意力结构、激活方式、上下文策略和模态接口。
2. 更大不等于更适合。总参数、激活参数、KV cache、吞吐和 tail latency 是不同维度。
3. 任务形状先于模型潮流。检索和分类常优先考虑 encoder-only；翻译与摘要常优先考虑 encoder-decoder；开放式对话与 Agent 多数时候才优先考虑 decoder-only。
4. **Dense** 与 **MoE** 是不同的失败模式。Dense 更容易预测；MoE 更容易扩容量，但要为路由和通信付出代价。
5. 长上下文必须同时看质量和服务。“支持 128K”不是结论，只是评测开始。
6. 多模态系统的难点在接口，不只在主干。编码器、投影器、cross-modal alignment 往往是故障源头。
7. 不要只读排行榜。技术报告、model card、配置文件和推理仓库，才是推断真实规格的依据。
8. **reasoning** 行为不只取决于 **backbone**。像 DeepSeek-R1 这样的例子提醒我们，后训练会显著改变模型表现。
9. 现代 **decoder** 配方的收敛是事实。RoPE、RMSNorm、SwiGLU、GQA 等组合之所以流行，不是因为“新”，而是因为它们在训练与推理上都更现实。
10. 最终问题永远是：这种架构在我的系统里会如何失败？只有能回答这个问题，选型才算完成。

对 AI 工程师来说，模型架构从来不是抽象理论。它直接决定训练成本、推理延迟、上下文能力、显存压力、部署方式，以及系统最后会如何失败。理解架构，本质上就是理解 **AI** 系统在生产中会表现成什么样。

14 参考资料

1. Vaswani et al. *Attention Is All You Need*. 2017.

2. Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019.

3. Raffel et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (T5)*. 2020.

4. Radford et al. *Improving Language Understanding by Generative Pre-Training (GPT)*. 2018.

5. Radford et al. *Language Models are Unsupervised Multitask Learners (GPT-2)*. 2019.

6. Brown et al. *Language Models are Few-Shot Learners (GPT-3)*. 2020.

7. Ouyang et al. *Training Language Models to Follow Instructions with Human Feedback (InstructGPT)*. 2022.

8. Chowdhery et al. *PaLM: Scaling Language Modeling with Pathways*. 2022.

9. Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023.

10. Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023.

11. Bai et al. *Qwen Technical Report*. 2023.

12. Qwen Team. *Qwen2 Release Notes*. 2024.

13. Qwen Team. *Qwen2.5 Release Notes*. 2024.

14. Jiang et al. *Mistral 7B*. 2023.

15. Jiang et al. *Mixtral of Experts*. 2024.

16. Du et al. *GLM: General Language Model Pretraining with Autoregressive Blank Infilling*. 2022.

17. DeepSeek-AI. *DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model*. 2024.

18. DeepSeek-AI. *DeepSeek-V3 Technical Report*. 2024.

19. DeepSeek-AI. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025.

20. Parmar et al. *Nemotron-4 340B Technical Report*. 2024.

21. OpenCompass. *A Universal Evaluation Platform for Foundation Models*. 2023.

22. Gao et al. *A Framework for Few-Shot Language Model Evaluation (lm-evaluation-harness)*. 2023.

23. Liang et al. *Holistic Evaluation of Language Models (HELM)*. 2023.

24. Hugging Face. *Open LLM Leaderboard*. 2023.

25. Chiang et al. *Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference (LMSYS)*. 2024.

26. Papers with Code. *Machine Learning Research & Benchmarks*. 2019.

27. Hendrycks et al. *Measuring Massive Multitask Language Understanding (MMLU)*. 2021.

28. Rein et al. *GPQA: A Graduate-Level Google-Proof Q&A Benchmark*. 2023.

29. Suzgun et al. *Challenging BIG-Bench Tasks and Whether Chain-of-Thought Can Solve Them (BBH)*. 2022.

30. Cobbe et al. *Training Verifiers to Solve Math Word Problems (GSM8K)*. 2021.

31. Hendrycks et al. *Measuring Mathematical Problem Solving with the MATH Dataset*. 2021.

32. Chen et al. *Evaluating Large Language Models Trained on Code (HumanEval)*. 2021.

33. Austin et al. *Program Synthesis with Large Language Models (MBPP)*. 2021.

34. Jimenez et al. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?*. 2024.

35. Zheng et al. *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. 2023.

36. Dubois et al. *AlpacaFarm / AlpacaEval: An Automatic Evaluator for Instruction-Following Models*. 2023.

37. Lin, Hilton, Evans. *TruthfulQA: Measuring How Models Mimic Human Falsehoods*. 2022.

38. Shi et al. *Language Models are Multilingual Chain-of-Thought Reasoners (MGSM)*. 2022.

39. Clark et al. *TyDi QA: A Benchmark for Information-Seeking Question Answering in Typologically Diverse Languages*. 2020.

40. Artetxe, Ruder, Yogatama. *On the Cross-lingual Transferability of Monolingual Representations (XQuAD)*. 2020.

41. Bai et al. *LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding*. 2023.

42. Kamradt. *Needle in a Haystack — Pressure Testing LLMs*. 2023.

43. Yue et al. *MMMU: A Massive Multi-discipline Multimodal Understanding and Reasoning Benchmark*. 2024.

44. Lu et al. *MathVista: Evaluating Mathematical Reasoning of Foundation Models in Visual Contexts*. 2023.

45. Liu et al. *MMBench: Is Your Multi-modal Model an All-around Player?*. 2023.