5. Common Models

Table of contents

aiengineeringhandbook.com

1 Overview: Why Model Architecture Matters

Imagine an enterprise search team. They see a 70B chat model near the top of public leaderboards and plug it straight into document ranking. The result is not dramatic, but it is fatal enough: relevance does not improve in any obvious way, latency rises, GPU cost balloons, batch throughput drops, and the online system becomes harder to debug. Later they switch the ranking layer to an encoder-only reranker and reserve the decoder-only model only for final answer synthesis. Quality becomes more stable, and cost drops sharply. The problem

aiengineeringhandbook.com

is not hyperparameters. It is **architecture fit**.

That is where this chapter begins. Model architecture is not just a paper label, and not just a brand on a leaderboard. It determines:

- whether a model is better at understanding or generation;
- whether training cost is dominated by parameter count, routing, or long sequences;
- whether inference latency is dominated by serial decoding, KV cache, or expert communication;
- whether long context is a marketable spec or a serviceable capability;
- whether multimodal inputs are aligned robustly or merely seem to work in demos.

> **!** Important
>
> This section first answers a few key questions:
>
> 1. Into which stable architectural prototypes can mainstream language models really be reduced?
> 2. Why do so many models with different names still share similar engineering recipes underneath?
> 3. Why can leaderboard scores never substitute for architectural understanding and application-specific validation when you evaluate a model?

From an engineering perspective, choosing an architecture is fundamentally a constrained optimization problem. You can write it as:

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

Here $\mathcal{A}$ is the architecture choice, Err is task error, Latency is end-to-end latency, Cost is training and inference cost, and Risk is system instability, failures, and safety risk. Different architectures do not merely move you along the same objective surface. They change the shape of the cost function itself.

So the question this chapter answers is not "which model is best," but something more stable, and much closer to production:

How do different model architectures actually shape an AI system's capabilities, cost, and failure modes?

This chapter is not about "which model is best." It is about **how mainstream models diverge architecturally, why they diverge, and what those differences mean in engineering practice**. Many model names look like a chain of brands, but strip away the marketing and they still collapse into a few stable prototypes: encoder-only, decoder-only, encoder-decoder, PrefixLM, plus variations layered on top of those skeletons—MoE, long context, local attention, multimodality, and more.

aiengineeringhandbook.com

It also answers a more important engineering question: **how should we evaluate these models, instead of merely reciting leaderboards**. Model selection is not about a single score. It is about whether the architecture, training objective, inference cost, evaluation setup, and target product scenario line up.

After this chapter, you should be able to:

- explain the main Transformer prototypes (encoder-only, decoder-only, encoder-decoder, PrefixLM) and where each fits;
- read core formulas such as attention, training objectives, KV cache, RoPE, and MoE routing, and know what they imply in engineering terms;
- understand the most common inference-time trade-offs: MHA / MQA / GQA, KV cache size, and the relation between context length and latency;
- find and verify a model's **actual specs** from technical reports, model cards, and configuration files;
- design a small but effective evaluation set, and avoid the most common benchmarking traps.

2 Architectural Prototypes: Four Basic Skeletons

> **!** Important
>
> This section first answers a few key questions:
>
> 1. What are the structural differences among encoder-only, decoder-only, encoder-decoder, and PrefixLM?
> 2. Why have most general-purpose chat models today converged on decoder-only?
> 3. When should you deliberately choose BERT or T5 instead of defaulting to a chat LLM?

Strip away the brand names and most mainstream language models reduce to four prototypes. The difference is not whether they are Transformers. It is **how information flows**.

- **Encoder-only** is more like a representation learner, and fits classification, retrieval, ranking, and embeddings;
- **Decoder-only** is more like a general-purpose generator, and fits open-ended generation, dialogue, code, and tool use;
- **Encoder-decoder** is more like a conditional generator, and fits translation, summarization, rewriting, and structured transformation;
- **PrefixLM** tries to fit "read the context fully" and "keep generation consistent" into one masking framework.

Why do most general-purpose LLMs today choose decoder-only? Because it

aiengineeringhandbook.com

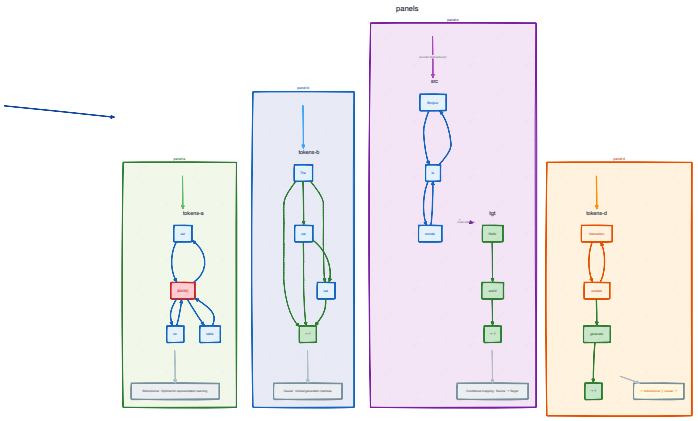

Figure 1: Information flow across different architectures: they are all Transformers, but the direction of information flow changes everything

unifies the training objective, data organization, alignment path, and inference interface into one pattern: given a prefix, keep generating. That unity is extremely helpful for large-scale pretraining and deployment. But it does not make other architectures disappear. If the task is retrieval and ranking, encoder-style models are often still the better fit; if the task is translation and summarization, seq2seq-style models are still very natural.

3 Math and Mechanisms at a Glance

> **!** Important
>
> Before comparing model families, fix a few mechanisms that determine production cost: attention, KV cache, positional encoding, MoE routing, and multimodal interfaces. Most of the real system cost grows out of these.

This section is not here so you can memorize formulas. It is here to build a stable engineering intuition: **why some models cost more, why some are better at generation, and why long context and MoE can drive system complexity up so quickly.**

3.1 Attention

The core of the Transformer is not "a deeper network." It is that information exchange among tokens is written explicitly as a layer of dynamic routing. Given

aiengineeringhandbook.com

input representations $X \in \mathbb{R}^{n \times d}$, a standard attention head is:

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

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

Here M is the mask. In decoder-only models, the most common case is the **causal mask**: the current position can read only itself and tokens to the left, never future tokens. Without that constraint, the training objective for autoregressive generation breaks.

Why divide by $\sqrt{d_k}$? Because in high dimensions, the variance of the dot product rises with dimension. Without scaling, softmax is more likely to become too sharp, gradients degrade, and training becomes less stable. The scaling does not change the relative ranking of "which token is more relevant," but it does change the numerical conditioning.

What gets expensive in practice is the complexity. Standard global self-attention forms an $n \times n$ score matrix, so:

- time and memory grow roughly as $O(n^2)$;
- if hidden dimension is d, total compute is often approximated as $O(n^2 d)$;
- if you materialize the score matrix directly, the intermediate tensor for one head alone has n^2 elements.

One often-missed order of magnitude is this: when $n = 131{,}072$ (that is, 128K), the attention score matrix for a single head already contains

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

elements. Stored in bf16, at 2 bytes each, **the score matrix for a single head alone** is about 32 GiB. Real systems do not materialize it that way. That is exactly why kernel optimizations such as FlashAttention have gone from "acceleration tricks" to "requirements."

3.2 KV Cache: Why Decoder-Only Gets More Expensive the Longer the Conversation Runs

In decoder-only inference, the model generates one new token at a time. At generation step t, the K and V for the previous $t-1$ tokens do not change. If you recompute the key/value states for the entire history at every step, the cost becomes too high to serve online. So inference systems store the historical K and V: the **KV cache**.

The cache size for one layer can be written roughly as:

aiengineeringhandbook.com

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

where:

- B is batch size;
- n is the current sequence length;
- G is the number of KV groups;
- d_k is the dimension of each KV head;
- b is the number of bytes per element;
- the leading 2 comes from storing both K and V.

If the model has L layers, total cache is approximately:

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

A numerical example: how far apart are 32K and 128K in production

Assume a decoder-only model with the following configuration:

- $L = 80$ layers;
- $G = 8$ KV groups;
- $d_k = 128$;
- bf16, so $b = 2$ bytes;
- start with a single request, so $B = 1$.

When $n = 32{,}768$ (32K):

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

Multiply by 80 layers, and total KV cache is about:

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

When $n = 131{,}072$ (128K):

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

Multiply by 80 layers, and total KV cache is about:

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

That is still only **batch size = 1**. If $B = 8$, the KV cache alone would theoretically approach 320 GiB. In other words, many models that "support 128K" are not failing because they cannot compute it. They fail because **in real concurrency, GPU memory dies first**.

aiengineeringhandbook.com

3.3 MHA, MQA, and GQA: Why Attention-Head Design Determines Serving Cost

Let the number of query heads be H, and the number of KV groups be G:

- **MHA**: $G = H$. Every query head has its own K, V. Strong expressivity, but the largest cache.
- **MQA**: $G = 1$. All query heads share one set of K, V. Smallest cache and lightest decode, but with some possible quality loss.
- **GQA**: $1 < G < H$. Multiple query heads share one K, V group. It is the compromise between quality and cache.

The KV cache is roughly proportional to G. So the immediate gain from moving from MHA to GQA, then to MQA, is not "fewer parameters." It is **less cache per request at decode time**, which makes it easier for the system to sustain long context and higher concurrency.

In serving systems, the bandwidth pressure of single-token decode can be understood roughly as:

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

So once n gets long, **HBM bandwidth** becomes the bottleneck even if FLOPs are still under control. That is why GQA / MQA matter so much in real systems.

3.4 MLP / FFN: Why Attention Still Needs a Block of "Local Compute"

Attention routes information across tokens. FFN performs a nonlinear transformation at each token position. They do different jobs. You need both. Without FFN, the model can still exchange information, but its expressive power drops sharply.

A classic FFN is written as:

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

Modern LLMs often use the GLU / SwiGLU family:

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

Compared with a single activation path, it adds a gating path: one part of the features acts as content, another part as the modulation signal. In practice, SwiGLU often delivers better expressive efficiency under a similar budget, which is why many modern decoders converge on it.

aiengineeringhandbook.com

From a systems perspective, FFN has another very practical meaning: **it often accounts for a large share of a Transformer's parameters and FLOPs**. That is why many MoE designs sparsify the FFN first instead of turning attention layers into experts.

3.5 Three Common Training Objectives

Training objectives shape what kinds of tasks a model is naturally good at, and in turn constrain architecture choices.

- **MLM (masked language modeling)**: mask some tokens and let the model recover them from left and right context.
- **Causal LM**: predict the next token from left to right, that is, $p(x_t \mid x_{<t})$.
- **Seq2Seq**: encode input x first, then conditionally generate output y.

The corresponding losses are often written as:

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

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

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

Why do most modern general-purpose LLMs use Causal LM? Because it unifies the training objective and the inference interface into the same thing: **given a prefix, continue generating**. Pretraining, continued pretraining, instruction tuning, and the inference interface all scale along the same engineering path.

But that does not make MLM and Seq2Seq obsolete. The former is still excellent for representation learning, retrieval, and classification. The latter is still the natural choice for translation, summarization, rewriting, and structured extraction.

3.6 Positional Encoding: Why Context Length Is Not Free

The Transformer itself does not know token order, so positional information has to be injected explicitly. RoPE (rotary position embedding) is a very common choice in modern decoders. It applies a position-dependent rotation to Q and K:

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

aiengineeringhandbook.com

Intuitively, RoPE does not add a position number directly into the vector. It rotates representations at multiple frequencies so that attention scores naturally carry relative positional information.

The problem is that every positional scheme has limits. **Length extrapolation** means making the model keep working on contexts longer than what it saw in training. The math can still run. That does not mean the semantics remain stable. If training never saw that distance distribution, quality usually degrades at inference time. Users see "supports 128K." Engineers should ask whether it can still retrieve, cite, and reason at 128K.

3.7 MoE Routing: Why It Expands Capacity and Still Creates System Trouble

The core idea of MoE (Mixture-of-Experts) is simple: not every token needs to pass through the full large MLP. A router selects a small number of experts to activate. A common form is:

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

This buys you a large total parameter count without making single-token FLOPs grow linearly with total parameters.

But MoE introduces two quantities you have to care about in engineering practice.

First is **expert capacity**. If a batch has N tokens, E experts, each token selects top-k experts, and the capacity factor is c, then the approximate capacity of a single expert can be written as:

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

If some experts are selected by too many tokens, the tokens beyond capacity must be dropped, downgraded, or rerouted.

Second is **load-balancing loss**. A common form adds an auxiliary term that pushes the token fraction f_e and average routing probability p_e of each expert closer to uniform:

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

What matters most here is not the exact constant. It is the engineering fact: the problem with MoE is not merely "do experts exist," but "are experts used stably and evenly."

aiengineeringhandbook.com

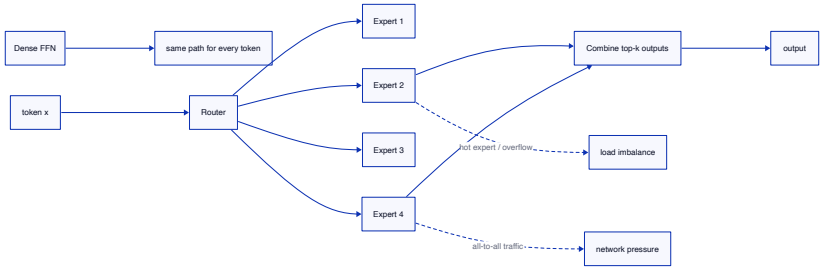

Figure 2

The cost of MoE does not disappear. It gets moved into **routing, load balancing, and communication**. In training, over-favored experts create overload and instability. In inference, the same issue appears as harder batching, worse tail latency, and heavier expert-parallel communication. MoE is not a "cheap big model." It is a big model that trades some dense compute cost for routing complexity.

4 Encoder-Only Models

> **!** Important
>
> This section first answers a few key questions:
> 1. What is an encoder-only model?
> 2. Why are BERT-like models still so strong at classification, retrieval, and embeddings?
> 3. When should engineers prefer them over chat-style LLMs?

The core advantage of encoder-only models is **compressing input into high-quality semantic representations**. They do not aim to keep writing the way chat models do. They aim to read the input, understand it, and produce a stable internal representation.

4.1 BERT

BERT is the representative architecture here. It uses bidirectional self-attention, so each token can read left and right context during encoding, then trains with the MLM objective to recover masked tokens. That objective naturally leans toward understanding and semantic compression, not left-to-right text continuation.

aiengineeringhandbook.com

From an engineering perspective, BERT-style models have held long-term value in three areas.

First, they are excellent for **retrieval and ranking**. Whether you are generating embeddings, using a dual-encoder for retrieval, or doing cross-encoder reranking, encoder-only models are usually higher-throughput, cheaper, and more stable than large decoder-only models.

Second, they fit **classification and discriminative tasks**. Content moderation, intent recognition, entity classification, ticket routing—these tasks do not need the model to write long answers. They need the model to separate semantic categories reliably. Forcing a generative model into this job usually buys you nothing but more latency and more cost.

Third, they work well as the **semantic infrastructure of a system**. In many AI products, the real traffic is not "generate a long answer," but continuously vectorizing documents, conversation snippets, code chunks, and queries. For that layer, BERT-like models are often the more realistic foundation.

Its limits are equally clear. BERT is not a native generator. It is not suited to open-ended continuation, dialogue, or tool use. You can of course attach a decoder on top of it, but that is no longer "using BERT directly for chat."

Engineering choice guidelines:

- when the output is a label, score, ranking, or vector, start with encoder-only;
- when throughput and unit cost matter more than "sounding human," start with encoder-only;
- when the task requires open-ended generation, multi-turn dialogue, or programmatic output, do not force encoder-only into the job.

5 Decoder-Only Models

> **!** Important
>
> This section first answers a few key questions:
> 1. Why did decoder-only become the mainstream of modern general-purpose LLMs?
> 2. Across families such as GPT, PaLM, LLaMA, Qwen, and Mistral, what is actually shared underneath?
> 3. What major system cost does decoder-only pay for that unity?

The core attraction of decoder-only is that it unifies training, alignment, and inference into one move: **given a prefix, generate the next token**. That makes it an excellent general-purpose language interface.

aiengineeringhandbook.com

5.1 Why It Dominates Modern LLMs

The victory of decoder-only did not come from one trick. It came from the unity of the whole engineering path.

- In pretraining, it learns next-token prediction;
- in instruction tuning, it is still continuing generation from a given context;
- in tool use, it is still generating structured call text;
- in dialogue systems, user messages, tool results, and system instructions can all be concatenated into one context prefix.

For large-scale training and product systems, that unity is enormously valuable. You do not need separate input/output shapes for different tasks, and you do not need to invent another interface on the inference side.

5.2 The GPT Family: The Victory of a Unified Generation Interface

The GPT line can be read, roughly, as an evolutionary chain that keeps strengthening the "general generation interface": **GPT → GPT-2 → GPT-3 → GPT-3.5 / InstructGPT → GPT-4**.

- **GPT-2** showed the industry that large-scale autoregressive pretraining could already produce obvious zero-shot and few-shot ability;
- **GPT-3** pushed scale far enough that prompt-based few-shot learning became a practical capability;
- **GPT-3.5 / InstructGPT** showed that "can continue text" is not enough; instruction tuning and RLHF are required to make the model act like a usable assistant;
- **GPT-4** presents itself more as a system-level and multimodal capability, but its internal architecture details have not been fully disclosed. That alone is an engineering reminder: **do not treat rumors about closed models as verifiable specs**.

The most important engineering lesson from the GPT family is not "parameter count keeps going up." It is that once the interface is unified around autoregressive generation, pretraining, alignment, tool use, and system integration can all keep evolving along the same backbone.

5.3 PaLM: A Systematized Recipe for Large-Scale Decoders

PaLM matters because it showed how large decoder-only models can evolve toward higher efficiency in a systematic way. One representative design choice in the report is to treat attention and FFN as **parallel Transformer sublayers** rather than unfolding them in a completely serial stack. Intuitively, this shortens the serial path and trims some communication cost.

aiengineeringhandbook.com

PaLM also adopted several design choices that later became very common: **SwiGLU, MQA, RoPE, and shared input/output embeddings**. None of them are there for show. Each is an engineering trade-off on FFN efficiency, KV cache control, positional modeling, or parameter redundancy.

The lesson from PaLM is that once models get large, what matters is no longer just "is it still a Transformer?" It is the data flow inside each block, the attention head configuration, the vocabulary design, and how friendly the implementation is to kernels.

5.4 LLaMA: The Progenitor of the Modern Open Decoder Recipe

LLaMA's value is less a single checkpoint than an **ecosystem progenitor**. LLaMA 1, LLaMA 2, and LLaMA 3 matter because they turned a very practical modern decoder recipe into the shared language of the open ecosystem: **RoPE, RMSNorm, SwiGLU, and later the more common GQA**.

This recipe is not dramatic, but it delivers stable real-world gains:

- training is more stable;
- inference cost is easier to control;
- the implementation is clean and reusable;
- the ecosystem can grow around it more easily.

That is why many later open models, despite different names, still feel unmistakably LLaMA-like underneath. When engineers read a model card, the first question should be whether the model inherits this modern decoder recipe—not what the marketing name is.

5.5 Qwen: Under the Same Skeleton, Data and Post-Training Create the Gap

The Qwen family makes one thing clear: **sharing a skeleton does not mean sharing capabilities.** Architecturally, Qwen also sits on the modern decoder-only path; for **Qwen2**, public materials emphasize **GQA** across different sizes to improve inference efficiency and KV cache usage. On context, pretraining in the family is typically built around **32K**, with some instruction variants extended to **128K**.

Qwen2.5 stays on the dense decoder path while pushing longer context, multilingual ability, code, and instruction following further. The thing engineers should remember is simple: even if two models both say "128K, decoder-only, GQA," they can still behave very differently on Chinese, code, tool use, and long-document QA. The gap often comes from data mix, tokenizer, post-training, and system integration—not just the block formula.

aienginneringhandbook.com

5.6 Mistral: Decoder Design with Efficiency First

Mistral's engineering signature is clear: it stays on the decoder-only mainline, but it actively optimizes inference and long-sequence cost. Representative choices include **GQA**, and in some models **sliding-window / local attention**. The gain is direct: lower compute and cache pressure, and better long-sequence throughput. The cost is just as direct: distant tokens no longer always interact globally.

That is why Mistral is often seen as the "efficiency-first" line. It does not abandon general generation. It asks a harder question instead: **if the general decoder form stays the same, where is cost surgery most worth doing?**

Within this family, **Mixtral** is the key extension, and we will unpack it in the MoE section. For now, remember one conclusion: Mistral represents efficiency optimization in dense decoders; Mixtral represents bringing sparse experts into the same design philosophy.

5.7 Why Decoder-Only Is Still Not the Universal Default Answer

Even though families such as GPT, PaLM, LLaMA, Qwen, and Mistral have all converged around decoder-only, its costs are just as stable:

- inference is serial generation, so end-to-end latency is constrained by design;
- prefill and KV cache costs rise quickly under long context;
- open-ended generation is harder to control in strictly structured formats;
- in online deployment, memory and cache management often become the bottleneck before parameter count does.

So decoder-only is the most general backbone today, but not the best answer for every task. **It wins on interface unity, not on being naturally cheapest for every job.**

6 Encoder-Decoder and Hybrid Models

> **!** Important
>
> This section first answers a few key questions:
>
> 1. Why does T5 use encoder-decoder instead of decoder-only like GPT?
> 2. Which tasks are better served by separating "understand the input" from "generate the output"?
> 3. Why is the PrefixLM route, as seen in GLM 4.5 / 4.6, worth understanding separately?

aiengineeringhandbook.com

6.1 T5: When the Task Is Naturally "Turn One Input into Another Output"

T5's core intuition is simple: when a task is fundamentally a conditional mapping, splitting input and output into two computation paths is often more natural. The encoder first reads the input in full and produces a stable representation; the decoder then uses cross-attention over that representation to generate the output.

This design is especially natural for translation, summarization, rewriting, question answering, and structured generation. Because the roles of input and output are explicit, the decoder does not have to juggle "understand the input" and "continue autoregressively" inside the same context.

Another important contribution of T5 is to write many tasks into a unified **text-to-text** interface. Whether the original task is classification, extraction, or summarization, the final form becomes "input string → output string." That makes supervision much more uniform and reduces fragmentation in task definition.

Its cost is equally clear. Compared with a decoder-only model of similar size, encoder-decoder usually has to run a full encoder and then do cross-attention during generation. So for open-ended chat and other tasks built around continuous generation, it is often less unified and less lightweight than decoder-only.

When should you prefer T5?

- When the task is a clear mapping from input to output;
- when the output structure is stable and does not require open-ended long dialogue;
- when you want stronger conditional generation, not a general-purpose assistant.

6.2 PrefixLM / GLM: A Compromise Between Understanding and Generation

The GLM line is not traditional encoder-decoder, but it answers a very similar question: **can the model read a given prefix more fully before it generates?**

In PrefixLM / GLM, the prefix is allowed more flexible context access, while the generation region still preserves causal consistency. So it is neither pure encoder-only representation learning nor pure left-to-right decoder-only generation. It opens a middle engineering path between the two.

This design is especially useful for infilling, blank filling, and conditional continuation with complex context. For engineers, the key to names like **GLM 4.5 / 4.6** is not to treat them as "just another model name," but to understand the

aiengineeringhandbook.com

masking philosophy underneath: in the same backbone, "read the context" and "generate the suffix" follow different rules.

Its cost is equally direct: the masking logic and training setup are more complex, and the inference stack is harder to unify than pure decoder-only. PrefixLM is a valuable middle path, but it will not become the universal product default as naturally as decoder-only.

7 Dense Models vs Mixture-of-Experts (MoE)

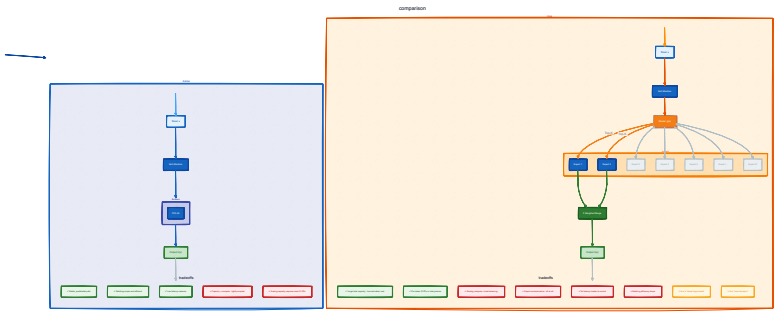

Figure 3: Dense vs MoE compute paths: the same capacity target, a different distribution of costs

> **!** Important
>
> This section first answers a few key questions:
>
> 1. What is the biggest engineering difference between dense models and MoE?
> 2. Why do models like Mixtral, DeepSeek-V2, and DeepSeek-V3 look like "huge total parameter count, but much smaller per-token activation"?
> 3. Why does MoE often create visible tension between offline evaluation and online deployment?
> 4. Why do average throughput and P99 often feel like two different systems when you serve MoE?

The advantage of a dense Transformer is that the path is simple and stable: every token goes through the full network, so compute behavior is easier to predict and batching is easier to manage. The cost is that capacity is tightly tied to FLOPs per token. If you want a larger model, you usually have to pay more compute almost linearly.

MoE tries to separate those two. Total parameters can be very large, but

aiengineering...1k.com

each token passes through only a few experts, so **total capacity** and **per-token activation cost** are no longer fully tied together. That is MoE's main engineering value.

Mixtral: why top-2 routing became a practical compromise

Mixtral is one of the clearest representative cases of mainstream MoE design today. In a configuration like 8x7B top-2, total parameters are large, but each token activates only two experts, so the number of activated parameters per token is far below the total parameter count.

That creates two immediate gains:

- You preserve high total capacity while controlling per-forward-pass cost;
- under some budgets, MoE becomes more realistic than a dense model with the same total parameter count.

But the cost is just as explicit. Routing itself takes compute. Expert assignment and communication get more complex. When different tokens choose different experts, batching and tail latency get harder to control. That is why Mixtral-like models often show a familiar online pattern: average latency looks fine, but P99 is ugly.

If you merge requests of different lengths and content into the same batch, the load from top-2 routing becomes highly input-dependent. **Average activated parameters** only describe the ideal compute per token. They **do not describe ideal service scheduling**. In real-time systems, routing variance is itself a cost.

DeepSeek-V2 / DeepSeek-V3: MoE is not just experts, but a redesign of cache and training objectives

DeepSeek-V2 and **DeepSeek-V3** deserve separate attention because they combine MoE with other system optimizations. In public material, V2 is described as **236B total parameters / 21B activated per token**, supports **128K** context, and introduces **MLA (Multi-head Latent Attention)** and **DeepSeekMoE**. V3 pushes further to **671B total parameters / 37B activated per token**, **128K** context, and adds **MTP (Multi-Token Prediction)** plus load balancing without an auxiliary loss.

What matters here is not memorizing the numbers, but reading the engineering intent underneath:

- **MoE** addresses capacity;
- **MLA** compresses KV cache further;
- **MTP** densifies training supervision and creates an interface for more aggressive inference acceleration;
- more complex load-balancing design shows that the bottleneck in MoE is not merely "do experts exist," but "can experts be used evenly and stably."

aiengineeringhandbook.com

In other words, DeepSeek V2 / V3 are not a pile of buzzwords. They are answering one coherent question: **how do you push capacity, context, and inference efficiency upward at the same time without letting per-token cost blow up?**

DeepSeek-R1-Zero / DeepSeek-R1: architecture is not everything; post-training reshapes reasoning behavior too

The value of **DeepSeek-R1-Zero** and **DeepSeek-R1** comes more from post-training than from the backbone itself. Public material says R1-Zero showed that reinforcement learning alone, without a traditional SFT cold start, can still induce strong reasoning behavior. R1 then added cold-start data to improve readability and product friendliness, and is built on **DeepSeek-V3-Base**.

That gives engineers an important reminder: **architecture sets the cost structure and the ceiling on capability, but reasoning style, usability, and alignment quality are often also shaped by post-training**. So when you look at a "reasoning model," you cannot stare only at the backbone name.

A standard MoE failure template: how routing collapse happens

A typical online failure looks like this:

- **Symptom**: one or two experts stay hot, all-to-all communication rises, P99 gets worse, but average throughput and offline benchmarks still look fine;
- **Root cause**: real traffic distribution differs from training distribution, and the router becomes highly biased on certain input types, which leads to very uneven expert utilization;
- **Mitigation**: stronger load-balancing constraints, more conservative capacity factors, router-temperature tuning, expert-parallel topology optimization, and dedicated monitoring for "hot experts."

This kind of failure makes the point: MoE capacity is not free. **It turns the problem of "a larger model" from matrix multiplication into matrix multiplication plus scheduling.**

A very practical rule

If your first priority is:

- Stable execution paths, easier deployment, and more predictable latency, prefer dense;
- the largest possible capacity, and you are willing to pay for routing and system complexity, consider MoE.

MoE is not a "more advanced Transformer." It is just another way to trade capacity against cost. The real engineering question is: **is your system more afraid of running out of compute, or of unstable routing?**

aiengineeringhandbook.com

8 Long-Context Architectures

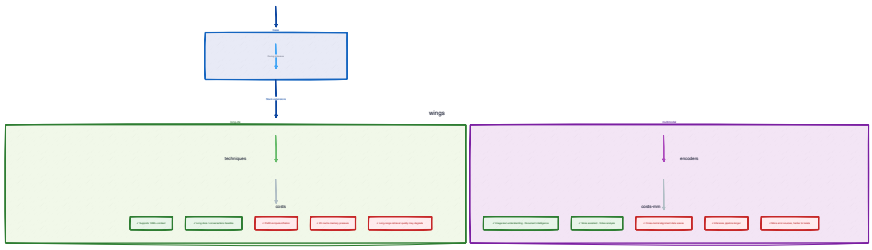

Figure 4: Long context and multimodality: extension layers stacked on top of the base architecture

> **!** Important
>
> This section first answers a few key questions:
>
> 1. Why is "supports long context" both a modeling problem and a systems problem?
> 2. What do sparse attention, local windows, cache compression, and positional extrapolation each actually solve?
> 3. Why do many models support 128K in config, but not with the same reliability at the semantic level?
> 4. How should you calculate the real serving budget of long context, instead of just reading the model card?

Long context is one of the most heavily marketed—and most misunderstood—architectural themes of the past two years. Users see a bigger window number. Engineers face three problems at once:

- attention prefill gets heavier;
- KV cache grows rapidly;
- positional generalization and retrieval quality start to degrade.

That is, long context is not one feature. It is a coupled problem of cost and capability.

How request-level cost grows

The end-to-end latency of one long-context request can be written roughly as:

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

where:

aiengineeringhandbook.com

- n_{in} is input length;
- n_{out} is output length;
- T_{prefill} is driven mainly by the quadratic complexity of attention;
- T_{decode} is more often dominated by KV reads and memory bandwidth.

That means pulling the context from 32K to 128K does not just make prefill heavier. **Every subsequent output token also decodes against a longer history.**

A numerical example: why 128K is not "just 4x 32K"

Keep using the previous configuration:

- $L = 80$;
- $G = 8$;
- $d_k = 128$;
- bf16, so $b = 2$;
- start with a single request, so $B = 1$.

As shown earlier, 32K gives total KV of about 10 GiB, while 128K gives about 40 GiB. That looks like a simple 4x linear increase, but in serving it usually triggers a chain reaction:

1. **Less free memory remains per request**, so concurrency falls;
2. **batching becomes harder**, because mixing short and long requests amplifies the tail;
3. **prefill time rises faster**, so users feel a much worse first-token latency;
4. **cache and weights compete for bandwidth**, which drags down decode throughput too.

So when teams move from 32K to 128K for the first time, a familiar pattern appears: offline single-request tests "work," but p95/p99 degrades sharply in production.

Path 1: keep dense attention, but make the implementation more serviceable

This is the most conservative and most common path. The model keeps global dense attention, but optimizes kernels, cache paging, and positional extrapolation—FlashAttention, PagedAttention, and various RoPE scaling techniques. The upside is an unchanged backbone and a friendly ecosystem. The downside is that quadratic complexity does not disappear; it is just implemented more efficiently.

This route is common in many modern decoder-only families, including many models in the **LLaMA** line, and models such as **Qwen2 / Qwen2.5**, which push pretraining context farther and then extend some instruction versions further still.

Path 2: change the attention connectivity graph

aiengineeringhandbook.com

Another class of methods changes which tokens can directly see which others. In some **Mistral** models, for example, sliding-window / local attention means tokens focus mainly on a local window instead of always attending to the entire history. The gain is lower partial compute and cache cost. The cost is that long-range dependencies no longer always have a direct path.

This design fits tasks where local patterns dominate. But if your product depends heavily on precise citation across long documents, long-range retrieval, and global consistency, local windows expose the weakness quickly.

Path 3: compress the cache representation more aggressively

MLA in **DeepSeek-V2 / V3** is a good example. Unlike GQA, which reduces cache by cutting the number of KV groups, MLA compresses the KV representation into a latent space and reconstructs it when needed. The gain is lower cache cost for serving long context. The cost is more complex implementation, training, and inference kernels.

This makes it clear that long-context architecture is not only about the attention graph. You can also attack the problem at the cache representation. In many systems, the real bottleneck is not parameter count. It is whether the cache can survive at all.

Path 4: train natively on long context instead of relying only on extrapolation

Some models train directly on longer sequences so they truly see longer distance distributions, instead of depending only on RoPE scaling to force a longer window. From an engineering standpoint, this is the most expensive path, but usually the most reliable. Its real value is not a larger window number. It is less degradation in long-range retrieval, citation, and reasoning.

In long-context systems, the first thing to break is often not parameter count but **memory and bandwidth**. Prefill throughput, KV cache, concurrency scheduling, prefix sharing, and batch packing often matter more to product experience than "is the model 70B?" That is why the usability of a long-context model requires two answers at once:

1. Can the math accept an input this long?
2. Can production serve an input this long stably at the target latency?

If your task is fundamentally "find a small number of relevant facts in a large corpus," do retrieval first, then consider pure long context. Turning a retrieval problem into a long-window problem usually amplifies both cost and error.

aiengineeringhandbook.com

9 Multimodal Architectures

> **!** Important
>
> This section first answers a few key questions:
> 1. A multimodal model is not just "one more input type." So what is actually changing?
> 2. How are image and audio encoders usually integrated into language models?
> 3. Why are cross-modal alignment errors often harder to localize than single-modal language errors?
> 4. Why can a system with correct-looking OCR still fail at visual-text reasoning?

Multimodal architecture solves a basic problem: **language models naturally consume tokens, but inputs from the real world are not naturally tokens.** Images, audio, video, layouts, and tables all have to be encoded first, then connected cleanly into the language backbone.

The most common mathematical form looks roughly like this:

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

where E_v is a vision or audio encoder, x_v is the raw modality input, W_p is a projector or adapter, and h_v is a "pseudo-token" representation the language backbone can consume.

Systems then usually follow one of two routes:

- **Concatenative fusion**: concatenate h_v before or after text tokens, and let a decoder-only backbone process them together;
- **Cross-attention fusion**: let language tokens read visual representations through cross-attention during generation:

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

Architecturally, the real change in multimodality is not "the input box now accepts images." It is whether **the backbone has a stable cross-modal interface and whether training has learned to align that interface to language tasks**.

1. Modality encoder + projector + language model

This is the most common system shape. A vision or audio encoder first converts the raw signal into continuous representations, then a linear projection or small adapter maps those representations into the hidden space the language model

aiengineeringhandbook.com

can consume. The benefit is that you can reuse a mature language backbone. The problem is that cross-modal alignment quality often depends almost entirely on whether this bridge layer is good enough.

This route is especially common in document understanding: page images go through a visual encoder, layout or OCR features are projected into language space, and the LLM then generates answers, extractions, or explanations.

2. A multimodal generator with cross-attention

Another route adds cross-attention explicitly, so the language decoder reads representations from an image or audio encoder while generating. Structurally, this is closer to attaching a vision encoder to a text decoder. It fits conditional generation tasks well, but the inference path is longer and training depends more heavily on high-quality alignment data.

This structure is natural for image captioning, visual question answering, video summarization, and speech-conditioned generation, because vision and audio are treated explicitly as the condition.

3. More unified tokenization or a shared backbone

A more aggressive route tries to unify different modalities into the same token format, or something close enough that one Transformer backbone can process them all. In theory this is elegant. In engineering it has the highest bar, because different modalities have very different statistical structures and time scales.

The upside is a unified backbone, a unified training objective, and a unified inference interface. The downside is that a unified representation does not necessarily mean unified usability. Many modalities are not naturally suited to being treated like language tokens.

> **Note**
>
> **Case study**
> Many teams run into the same confusion the first time they build a multimodal document system:
> - The OCR text is almost entirely correct;
> - the model can paraphrase the general content of the page;
> - but once the question becomes "what amount is jointly determined by the third column of the table and the condition in the footer?", the error rate jumps.
>
> The root cause is usually not "the model cannot read words." It is that **visual features and language instructions were never aligned stably**. Possible failure points include:
> - the projector flattens local visual relationships too aggressively and loses spatial structure;
> - the model sees OCR-concatenated text instead of layout relation-

aiengineerinhandbook.com

ships;
- the training data mainly rewards "understand local images," not "reason jointly across regions."

That is why document understanding systems often need extra 2D positional encodings, region tokens, structured table extraction, or even a dedicated layout encoder. **Multimodal systems often fail not because they did not see the signal, but because what they saw never aligned with the language task.**

From the perspective of this chapter, the important thing is not memorizing a visual-language model name. It is understanding that **the cost of multimodal architectures comes from the cross-modal interface, not the language backbone itself.**

It is worth noting that the public product form of **GPT-4** already shows clear multimodal system capability, but its detailed internal architecture has not been fully disclosed. Again, that is a reminder: for closed models, you can analyze public behavior and system interfaces, but you should not invent unverifiable internal implementations.

10 How to Evaluate Model Architectures

! Important

This section first answers a few key questions:

1. How should engineers compare architectures instead of staring only at leaderboards?
2. What problems are benchmarks, task evaluations, and efficiency metrics each solving?
3. Why can the same high-scoring model still be a bad fit for your product?

To evaluate an architecture, you need at least three layers in view:

1. **Capability benchmarks**: can the model do it at all?
2. **Efficiency metrics**: how much time, memory, and money does it take to do it?
3. **Task fit**: in your real scenario, does it fail in the right way?

The first two tell you what is possible in theory. The third tells you whether it is worth shipping.

10.1 Capability Benchmarks

This layer asks what the model can do on public benchmarks. For a general-purpose LLM, a practical minimum set should cover:

aiengineeringhandbook.com

- general knowledge and reasoning: MMLU / MMLU-Pro, GPQA, BBH;
- math: GSM8K, MATH;
- code: HumanEval, MBPP, SWE-bench;
- instruction following / dialogue: MT-Bench, AlpacaEval, LMSYS Arena;
- truthfulness / safety: TruthfulQA, plus red teaming;
- multilingual: MGSM, TyDiQA, XQuAD;
- long context: LongBench, Needle-in-a-haystack;
- multimodal: MMMU, MathVista, MMBench.

The value of these benchmarks is that they give you a shared coordinate system. Their limit is just as clear: the distribution is static, easy to optimize against, and often not your task.

10.1.1 Efficiency Metrics

Two models with the same accuracy can live on completely different production scales. Architecture evaluation has to include, explicitly:

- prefill throughput;
- decode throughput;
- P50 / P95 / P99 latency;
- memory usage per request;
- KV cache growth curve;
- batching efficiency and concurrency ceiling;
- MoE expert communication and tail latency;
- training tokens/sec and cluster scaling efficiency.

These are not after-the-fact deployment details. They are part of the architecture judgment itself. A model that is stronger on offline benchmarks but whose P99 is too high to ship is not the better engineering choice.

10.1.2 Task Fit

What usually decides whether an architecture is right is not the aggregate leaderboard, but questions like:

- Is your task closer to retrieval, translation, dialogue, or an agent?
- Are user inputs short Q&A, or long documents and long conversations?
- Are you more afraid of hallucination, latency, cost, or reasoning failure?
- Do you need images, audio, tables, or layout understanding?

That is why benchmarks are always only a starting point. Real selection should begin with product failure modes, then ask which architecture is best suited to absorb them.

10.1.3 Evaluation Methods

Evaluation methods can be divided roughly into four types. Each answers a different version of "is it good?"

aiengineeringhandbook.com

1. **Offline automated evaluation**
   Fast, cheap, and reproducible. Good for knowledge, math, code, and other tasks with clear answers or scoring rules. The downside is sensitivity to prompt format, data contamination, and benchmark overfitting.

2. **LLM-as-a-judge**
   Useful when you need to compare writing quality, instruction following, or completeness at scale, where automatic scoring is hard. The issues are positional bias, verbosity bias, style bias, and drift in the judge model itself. Common mitigations include randomizing candidate order, blinding model identity, pinning the judge version, and calibrating on a small human-labeled set.

3. **Human preference testing**
   When you truly care about user experience, tone, helpfulness, safety boundaries, and product satisfaction, human testing is still the most trustworthy signal. It is expensive, slow, and noisy unless sampling and annotation guidelines are designed carefully.

4. **System-level / agent-level evaluation**
   This looks at end-to-end task success: whether tools are called correctly, whether multi-step tasks finish, whether repository edits can be merged. It is closest to real value and hardest to standardize. Infrastructure cost is highest.

A common mistake is to treat an LLM judge as a "cheap replacement for humans." A more accurate view is this: **an LLM judge is a high-throughput approximate measurement tool, not the final truth.** When the task involves style preference, brand voice, medical or legal decisions, and other high-risk judgments, human evaluation is still irreplaceable.

10.2 Common Evaluation Pitfalls

Many conclusions of the form "model A is stronger than model B" are later defeated not by the models themselves, but by the evaluation setup. The most common traps are:

- **Inconsistent prompting**: small changes in 0-shot, few-shot, CoT, system prompt, or template format can all move scores;
- **inconsistent decoding**: temperature, top-p, fixed seed, and whether multiple samples are allowed all change results;
- **contamination**: overlap between train and test, memorized questions, and repeated optimization against public benchmarks;
- **inconsistent scoring**: exact match, F1, pass@k, LLM judge, and human preference measure different things and cannot be mixed naively.

The reason "MMLU is high, but real customer experience is mediocre" usually comes down to **distribution mismatch**. MMLU is closer to a clean multiple-choice knowledge test. Customer questions are multi-turn, ambiguous, context-

aiengineeringhandbook.com

dependent, tool- or retrieval-heavy, and constrained by style and safety. Doing well on a static, neatly formatted benchmark does not mean the model has adapted to real product noise and workflow.

10.3 Tools, Leaderboards, and First-Hand Sources

A mature evaluation workflow usually depends on three kinds of resources at the same time:

- **evaluation suites / harnesses**: OpenCompass, lm-evaluation-harness, HELM;
- **leaderboards / aggregators**: Hugging Face Open LLM Leaderboard, LMSYS Chatbot Arena, Papers with Code;
- **first-hand spec sources**: technical reports, model cards, configuration files, and inference repositories.

Evaluation harnesses solve **execution consistency**; leaderboards provide **a shared comparison frame**; first-hand specs provide **factual consistency**. You need all three.

10.4 Recommended Minimal Benchmark Set

A practical rule is simple: **use a small, diverse set instead of one giant aggregate leaderboard.** For a general-purpose LLM, the minimum should cover general knowledge / reasoning, math, code, instruction following / dialogue quality, truthfulness / safety, multilingual ability, and long context. If the model has visual ability, add multimodal evaluation on top.

The reason not to trust a single number is simple: different benchmarks use very different input formats, scoring methods, and capability targets. Flattening them into one sum usually washes out the real differences.

The two formulas that show up most often in this section are:

- Perplexity:

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

  Perplexity measures how uncertain the model is about the next token under a given distribution. It is useful for thinking about language modeling quality, but **it is not equivalent to** instruction following, alignment, or tool-use ability.

- pass@k in code: if you sample n programs from the model and c of them pass the tests, a common unbiased estimate is:

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

aiengineeringhandbook.com

It reflects the real usage pattern of "give the model multiple chances" better than pass@1, but it also mixes the gain from a larger sampling budget into what looks like model ability.

The benchmark set below is a practical starting point, not a final answer.

General knowledge and reasoning

- MMLU / MMLU-Pro
- GPQA (hard scientific reasoning)
- BBH (Big-Bench Hard)

This group sketches the baseline profile of a model's broad knowledge and general reasoning. **MMLU** is closer to a wide set of multiple-choice knowledge questions, which makes it useful as a cross-model baseline. **MMLU-Pro** is harder and less vulnerable to superficial template tricks, so its scores generally should not be read as directly comparable to the original MMLU.

GPQA adds hard scientific reasoning. It separates "can solve routine knowledge questions" from "can stay stable on unfamiliar, complex scientific ones." **BBH** covers a wider mix of hard tasks and helps you see the boundary across multiple reasoning styles.

Math

- GSM8K (elementary to middle-school math)
- MATH (competition-style math)

These two benchmarks operate at different difficulty levels. **GSM8K** is better for checking basic math reasoning over shorter chains. **MATH** exposes weaknesses faster in long reasoning chains, symbolic manipulation, and the stability of intermediate steps. Looking at only one of them makes it easy to misread a model's math boundary.

One often-missed variable in math evaluation is self-consistency or majority vote: both can significantly lift final accuracy. At that point, you have to be clear about what you are measuring: "single-shot output quality" or "problem-solving ability under a reasoning budget." In products, those imply very different cost structures.

Code

- HumanEval
- MBPP
- SWE-bench

In code, **pass@1** measures the probability that the model gets it right on the first output. **pass@k** measures the probability that at least one candidate passes the tests when the model gets k samples. The former is closer to single-shot autocomplete. The latter is closer to a workflow that lets the model try several times and then filters.

aiengineeringhandbook.com

HumanEval and MBPP are more static, function-level tasks; they are good for quickly checking the basics of code generation. **SWE-bench** is closer to real software repair: the model has to understand the issue in repository context, edit files, and pass tests. You need both views, because a model can be good at short functions and still fail at locating and fixing bugs in a real codebase.

Instruction following / dialogue quality

- MT-Bench
- AlpacaEval
- LMSYS Arena

This group asks a different question. Not "can the model solve the problem?" but "does it sound like a usable assistant?" **MT-Bench** and **AlpacaEval** are better for structured comparison of instruction following and answer quality. **LMSYS Arena** turns open-ended chat quality into relative rank through blinded human head-to-head comparisons.

Elo matters because dialogue quality is hard to exhaust with a static reference answer, and pairwise comparison is often more stable than absolute scoring. But Elo is not a product metric. For a customer-facing chatbot, internal evaluation should at least cover: **accuracy, safety, style consistency, refusal boundaries, latency, cost, and tool-call success rate**.

Truthfulness / safety

- TruthfulQA
- Safety red teaming: define a clear policy / threat model first, then combine automated scanning with manually constructed data; example tools include garak and AEGIS, referenced in the NVIDIA Nemotron-4 model card

TruthfulQA is not about ordinary factual QA. It asks whether the model will echo common misconceptions, superstitions, or false priors. In other words, it measures a very specific failure mode that matters a lot in real products: the model repeats popular falsehoods because it wants to sound like an answer.

Safety evaluation cannot rely on one dataset either. First define the threat model: are you worried about jailbreaks, hateful content, self-harm advice, privacy leaks, or tool abuse? Then combine automated scanning, human red teaming, adversarial prompts, context injection, and system-level stress tests.

Multilingual

- MGSM
- TyDiQA
- XQuAD

The point of multilingual evaluation is to test whether capability actually transfers across languages, not whether it only works in English. **MGSM** is useful for checking whether math reasoning stays stable cross-lingually. **TyDiQA**

aiengineeringhandbook.com

leans more toward multilingual QA and information understanding. **XQuAD** provides a cross-lingual reading comprehension comparison.

Why can you not skip this group? Because multilingual performance depends not just on scale, but on tokenizer design, corpus ratios, script type, and the language coverage of post-training. A model that is extremely strong in English can still fall onto a very different curve in Chinese long-form QA, Arabic search QA, or mixed code-and-natural-language scenarios.

Long context

- LongBench
- Needle-in-a-haystack: use controlled synthetic retrieval probes; one commonly cited public implementation is https://github.com/gkamradt/LLMTest_NeedleInAHaystack

Long-context evaluation must be split into at least two kinds. **Needle-in-a-haystack** asks whether the model can recover a key fact buried in long text at different positions. It is controlled and interpretable, so it is good for checking whether retrieval ability still exists. **LongBench** is closer to a collection of long-input tasks—summarization, QA, comprehension—that better resemble real usage.

You need both. Being able to retrieve the needle does not mean the model works reliably on long-document tasks. Getting a decent average on long tasks does not mean the model can locate evidence well. The most common mistake in long context is taking "the math can accept this length" for "the semantics are still usable at this length."

Multimodal

- MMMU
- MathVista
- MMBench

The key in multimodal evaluation is to separate "what did the model see?" from "what did it understand?" **MMMU** emphasizes cross-disciplinary knowledge and multimodal reasoning. **MathVista** stresses charts, geometry, and visual mathematical reasoning. **MMBench** gives a broader comparison of visual-language ability.

If you only test OCR or image captioning, you learn whether the model can turn images into text. You do not learn whether it can reason, compare, calculate, and make task decisions around those images. A real multimodal system should close the loop among visual perception, text understanding, cross-modal alignment, and downstream task completion. The evaluation has to cover those layers too.

aiengineeringhandbook.com

> **!** Important
>
> **Why benchmarks are not enough to replace architectural understanding**
> Because benchmarks mostly answer "how does it perform on this set of problems?", while architectural understanding answers "why does it behave that way, and where will it break after deployment?"
> A model can score highly on MMLU, GPQA, or Arena and still fail to imply any of the following:
> - that it fits long-session customer support;
> - that its KV cache can support your concurrency target;
> - that it remains stable on your Chinese tickets, code repositories, or multimodal documents;
> - that its P99 latency fits your serving budget.
>
> What engineers actually need is not an abstract conclusion that "this model is stronger." It is the ability to **map architectural traits onto system constraints**.

11 What Architecture Actually Changes

> **!** Important
>
> This section first answers three questions:
> 1. Why is architecture not just a theoretical difference, but a behavioral difference in systems?
> 2. Through which paths does one architecture choice affect training and inference?
> 3. Why are "strong capability" and "deployable capability" so often constrained by different things?

From a systems perspective, model architecture mainly changes five things.

First, **the direction of information flow**. A bidirectional encoder lets each token read both left and right context, which makes it better at understanding and semantic compression. A causal decoder can read only the left, which makes it better at continuous generation. Encoder-decoder splits "read the input" and "write the output" into two computation paths.

Second, **how parameters get activated**. In a dense model, every token passes through the whole network. In MoE, only a few experts activate. The former has a stable execution path; the latter trades routing complexity for model capacity.

Third, **how state grows with sequence length**. Long context is not just "more tokens fit inside." It means heavier prefill, larger KV cache, higher

aiengineeringhandbook.com

memory-bandwidth pressure, and concurrency behavior that is harder to control.

Fourth, **how external modalities enter the system**. A multimodal model is not just "an extra image input box." Images, audio, and video have to be encoded into representations the language backbone can consume, and cross-modal alignment has to hold in both training and inference.

Fifth, **where system bottlenecks appear**. Some architectures are hungrier for training compute, some for inference memory, some for all-to-all communication, some for positional extrapolation and cache management. In practice, choosing an architecture means choosing your dominant bottleneck.

Architecture choice	What it changes	Training impact	Inference / deployment impact	Systems it fits best
Encoder-only	Representation learning and bidirectional context	Better suited to discriminative and semantic-representation tasks	One-pass encoding, high throughput	Retrieval, classification, ranking, embeddings
Decoder-only	Autoregressive generation ability	Training objective and inference interface are highly unified	Serial decoding, KV cache grows with length	Dialogue, code, agents, general generation
Encoder-decoder	Conditional generation mapping	Clear input/output roles, natural supervision	Must maintain both encoder and decoder compute	Translation, summarization, rewriting, structured output
PrefixLM / hybrid masking	A compromise between understanding and generation	More flexible masking design, but more complex	Inference and implementation logic are harder to unify	Infilling, conditional continuation, complex prefix reading
MoE	Activated parameter count	Larger capacity, but routing is harder to train	Less stable latency, more complex communication	Large high-capacity models

aiengineeringbook.com

Architecture choice	What it changes	Training impact	Inference / deployment impact	Systems it fits best
Long-context mechanisms	Sequence length and long-range dependencies	Longer-sequence training and positional generalization are harder	Higher prefill and KV-cache pressure	Long documents, long conversations, repository-scale tasks
Multimodal integration	Input types and alignment method	Depends more heavily on cross-modal alignment data	Longer inference path, more sources of error	Vision understanding, voice assistants, document intelligence

12 Engineering Case Studies

> **!** Important
>
> What follows is not a postmortem from one single company, but a set of composite failure modes that appear across the industry again and again. Each case answers three questions: what the symptom is, what the root cause is, and what the engineering lesson is.

12.1 Taking a Chat Model and Using It as a Reranker: Quality Did Not Improve, Cost Exploded First

Symptom: the team used a large chat model to score search results one by one. Per-request latency rose sharply, GPU cost ballooned, and the ranking was still not more stable than a strong encoder-only reranker.

Root cause: architecture mismatch. Ranking is fundamentally a representation and discrimination problem. It wants stable relevance scores and high throughput. A decoder-only model is doing serial generative inference instead: slower, and harder to make score consistently.

Engineering lesson: architecture should follow the **output shape**, not leaderboard heat. If the output is a vector, label, or score, prefer encoder-only. Open-ended text is where decoder-only should lead.

aiengineeringhandbook.com

12.2 MoE Looks Strong Offline, but Online Tail Latency Goes Out of Control

Symptom: offline benchmarks look excellent, average decode throughput is not bad, but under high concurrency in production, P99 shoots up, batching efficiency falls, and GPU utilization is not nearly as high as expected.

Root cause: MoE routing sends different tokens to different experts, which creates uneven expert load, more complex communication, and requests that are harder to pack cleanly at the tail. The averages look fine, but tail latency is amplified by routing complexity.

Engineering lesson: when evaluating MoE, do not look only at "activated parameters are cheaper." You also have to evaluate routing balance, expert-parallel communication, batching strategy, and P99. For many real-time systems, **predictability** is itself a capability.

12.3 128K Is on the Model Card, but the System Dies at 128K

Symptom: the team expands the context window from 32K to 128K, hoping to reduce retrieval and summarization pipelines. First-token latency doubles, concurrency falls, and long-document QA still frequently misses critical facts buried in the middle.

Root cause: the larger window raises system cost through prefill and KV cache, while the model itself has not learned stable retrieval and citation behavior at that same length. So "the math can hold 128K" never becomes "the semantics work better at 128K."

Engineering lesson: long context raises both capability and cost. It is never free. Before launch, you need both long-context quality evaluation and serving stress tests. The maximum window number alone means almost nothing.

12.4 A Top-Leaderboard Model Still Fails in Real Customer-Service Tickets

Symptom: the model ranks highly on MMLU and Arena, but once it enters a real customer-service workflow, it often answers the wrong question, fails to cite internal policy, and has a high tool-call failure rate.

Root cause: public benchmarks and real ticket distributions are very different. The former are clean problems or generic conversations. The latter are multi-turn, noisy, constrained by system rules, and often require retrieval and tool use. The architecture may be strong, but the product pipeline was never validated around real failure modes.

Engineering lesson: leaderboards are good for shortlisting candidates, not for launch decisions. Real product evaluation has to be designed around the task,

aiengineeringhandbook.com

tools, context, and style constraints.

12.5 The Multimodal Demo Is Impressive, but Document Understanding Is Not Reliable

Symptom: the model looks strong on image-QA demos, but once it moves into invoices, screenshots, tables, and layout-heavy documents, the error rate rises sharply—especially on cross-region citation and joint visual-text reasoning.

Root cause: the weak point in multimodal systems is often not the language backbone, but the modality encoder, the projection layer, or cross-modal alignment. The model "saw something," but that does not mean it can align those visual signals stably to a language task in a structured layout.

Engineering lesson: when evaluating multimodal architectures, do not stop at OCR or simple image captioning. You have to cover layout understanding, cross-region citation, visual reasoning, and end-to-end task completion.

13 Chapter Summary

A practical way to approach model selection is to answer four questions first:

1. **What is the output shape?** A vector, score, label, or open-ended text?
2. **What is the dominant bottleneck?** Training compute, inference memory, end-to-end latency, or engineering complexity?
3. **How long is the context?** Do you need stable 4K serving or 128K offline analysis?
4. **Do you need external modalities?** Is it just text, or do images, audio, tables, and layout enter the system too?

If you zoom out one level further, architecture selection is really about minimizing damage along five dimensions:

- quality ceiling;
- latency and throughput;
- memory / bandwidth pressure;
- implementation and operations complexity;
- safety and debuggability.

aiengineeringhandbook.com

Architecture prototype	Representative models / families	Dominant advantage	Dominant cost	When to prefer it	Common misuse
Encoder-only	BERT	Strong representation learning, high throughput, low latency	Poor at open-ended generation	Retrieval, classification, reranking, embeddings	Using it for long dialogue or open-ended writing
Decoder-only	GPT, PaLM, LLaMA, Qwen, Mistral	Unified interface, strong generation, best fit for general assistants	Serial decoding, KV cache grows with length	Dialogue, code, agents, free-form generation	Using it to replace every discriminative task
Encoder-decoder	T5	Clear input/output roles, natural conditional generation	Longer inference path, more complex deployment	Translation, summarization, rewriting, structured extraction	Treating it as the default chat backbone
PrefixLM / Hybrid	GLM 4.5 / 4.6	A fine-grained compromise between understanding and generation	More complex masking logic and inference stack	Infilling, complex conditional continuation	Assuming it naturally replaces encoder-decoder
Dense (MoE-free)	Dense LLaMA / Qwen / Mistral	Stable execution path, high predictability	Capacity is tightly bound to per-token compute	Real-time systems, low tail-latency serving	Forcing a much larger dense model into a fixed budget

aiengineeringhandbook.com

Architecture prototype	Representative models / families	Dominant advantage	Dominant cost	When to prefer it	Common misuse
MoE	Mixtral, DeepSeek-V2, DeepSeek-V3	Expand total capacity at lower activation cost	Complex routing, load balancing, and communication	High-capacity inference, lowering large-model cost	Looking only at average latency and ignoring P99
Long-context variants	Long-window Qwen2/2.5, LLaMA-line models, Mistral windows, DeepSeek MLA	Can handle long documents and long conversations	Prefill, KV cache, and positional generalization all get harder	Long-document QA, repository-scale analysis	Assuming "larger window" means "better quality"
Multimodal	GPT-4-like product forms, VLMs, speech-language models	Richer input types	More alignment, projector, and cross-modal failure points	Visual QA, document understanding, voice assistants	Testing only OCR and not reasoning or grounding

A very practical rule of thumb is:

- **If the output is a vector or score, think encoder-only first;**
- **if the output is text and the interface must stay general, think decoder-only first;**
- **if the input-to-output mapping is explicit and stable, think encoder-decoder first;**
- **if capacity is the first priority and you can absorb routing complexity, only then think MoE;**
- **long context and multimodality are never free upgrades; they expose system bottlenecks earlier.**

13.1 Question Summary

1. What is model architecture, and how is it different from parameter count?

aiengineeringhandbook.com

Model architecture describes how the computation graph is organized: how information flows, how parameters activate, how context is read, and how outputs are generated. Parameter count is only the scale of that graph. It says nothing about path design.

2. Why did the Transformer become the default backbone of modern language models?
Because attention lets the model represent token-to-token relationships dynamically. It scales well, trains in parallel, and combines naturally with later designs such as long context, multimodality, and MoE.

3. What is the essential difference among encoder-only, decoder-only, encoder-decoder, and PrefixLM?
Encoder-only emphasizes bidirectional understanding and representation. Decoder-only emphasizes causal generation. Encoder-decoder splits input understanding and output generation into two paths. PrefixLM uses different masking rules inside one backbone for "read the prefix" and "generate the suffix."

4. Why do most general-purpose LLMs choose decoder-only?
Because next-token prediction unifies pretraining, instruction tuning, tool use, and the inference interface into one form. It is the cleanest engineering scaling path.

5. What constraints usually drive architecture design?
Training scalability, inference latency, memory, context length, modality integration, alignment method, and the shape of the target product task.

6. What is the core trade-off between dense models and MoE?
Dense models have stable execution paths and are easier to deploy, but capacity is tightly tied to compute. MoE expands capacity at lower per-token activation cost, but adds routing, load-balancing, and communication complexity.

7. Why do some models support longer context?
Because they make explicit changes in positional modeling, long-sequence training, attention graphs, or cache representation. Long context does not come from one config flag.

8. How does attention structure affect compute cost?
Global dense attention gives you $O(n^2)$ prefill cost. MHA / MQA / GQA affect KV cache. Local attention and cache compression change the serving-cost structure for long sequences.

9. Which architectural choices usually help reasoning?
A stronger backbone raises the ceiling, but reasoning behavior often also depends on training objectives, long-horizon supervision, and post-training. You have to look at architecture and post-training together.

10. Why are some models easier to extend to multimodality?
Because their backbone and interfaces more naturally accept representations

aiengineeringhandbook.com

from external encoders, or because they already support cross-attention or a unified token interface.

11. Why can a model claim 128K context and still fail on long-document QA?
Because "it can fit 128K tokens" only describes the mathematical window. It does not mean the model has learned to retrieve, locate, and cite facts reliably at that length. Meanwhile, the serving system may already be collapsing under prefill and KV cache.

12. How should you choose a model architecture in production systems?
Start with task shape and failure modes, then look at cost and latency constraints, and only then consult public leaderboards. If the task is mostly retrieval and classification, prefer encoder-only. If it is mostly open-ended generation, prefer decoder-only. If it is a stable conditional mapping, prefer encoder-decoder.

13. How do model size, activated parameters, KV cache, and latency relate?
Total parameter count determines loading and training scale. Activated parameters determine per-token compute cost. KV cache determines memory pressure under long context and concurrency. End-to-end latency is the combined effect of prefill, decode, and scheduling.

14. When should production systems prefer encoder-only over decoder-only?
When the output is a score, label, vector, or ranking rather than open-ended text. When throughput, cost, and stability matter more than "sounding human," encoder-only should usually come first.

15. Why do different architectures keep winning on different tasks?
Because different tasks require different information flow. Retrieval needs stable representations. Translation needs a clean input-output mapping. Dialogue needs a unified generation interface. Multimodality needs a stable bridge across modalities.

16. What new failure modes appear when you serve MoE models that dense models do not have?
Hot experts, capacity overflow, amplified all-to-all communication, batching mismatch, sharply worse P99, and routing collapse when traffic distribution changes.

17. What are the main trends in architecture evolution right now?
First, modern decoder recipes keep converging. Second, MoE and cache compression keep expanding capacity while controlling cost. Third, long context is moving from "window stretching by config" toward tighter coordination across training and serving. Fourth, multimodal interfaces are becoming a core part of backbone design.

aiengineeringhandbook.com

13.2 A Practical Checklist for AI Engineers

Figure 5: Model-selection matrix: task × architecture × cost — quick reference

Task shape	Default architecture to prefer	Why	Main risk
Semantic retrieval / reranking / classification	Encoder-only (BERT-like)	The output is a vector, score, or label; throughput matters more	Misusing it as an open-ended generator
General assistants / code completion / agents	Decoder-only (GPT / LLaMA / Qwen / Mistral-like)	Unified interface; best fit for continuous generation and tool use	KV cache, serial decode, long-session cost
Translation / summarization / rewriting / structured extraction	Encoder-decoder (T5-like)	Input and output roles are clear; conditional generation is natural	Longer inference path
Very large capacity under budget pressure	MoE (Mixtral / DeepSeek-V2 / V3-like)	Large total capacity with relatively controlled activation cost	Routing, communication, P99, and scheduling complexity
Long-document QA / long conversations	Long-context architecture + retrieval	You need both the window and retrieval	Expanding the window without budgeting for serving
Multimodal document understanding / voice assistants	Multimodal architecture	Vision or audio representations must enter the system stably	Alignment, projector, and grounding failures

In real product teams, architecture selection is rarely one definitive shot. The more realistic workflow is usually:

- first build the baseline with the cheapest, most stable architecture;
- then prove that the more complex architecture actually improves your task;

aiengineeringhandbook.com

- only then pay the training, serving, and operations cost for that gain.

That is why **architectural understanding is not "knowing what models exist," but knowing when not to upgrade.**

What this chapter is really trying to build is not "recognize a few model names," but a more stable engineering perspective.

1. **Architecture sets system constraints.** Do not treat the model as an abstract black box of "capability." Ask first about attention structure, activation pattern, context strategy, and modality interface.
2. **Bigger does not mean better suited.** Total parameters, activated parameters, KV cache, throughput, and tail latency are different dimensions.
3. **Task shape comes before model fashion.** Retrieval and classification often point first to encoder-only. Translation and summarization often point first to encoder-decoder. Open-ended dialogue and agents are where decoder-only usually comes first.
4. **Dense and MoE fail differently.** Dense is easier to predict. MoE expands capacity more easily, but you pay for routing and communication.
5. **Long context must be judged on both quality and serving.** "Supports 128K" is not a conclusion. It is the start of evaluation.
6. **The hard part of multimodal systems is the interface, not only the backbone.** Encoders, projectors, and cross-modal alignment are often where failures begin.
7. **Do not read only leaderboards.** Technical reports, model cards, configuration files, and inference repositories are what let you infer the actual specs.
8. **Reasoning behavior does not depend only on the backbone.** Examples like DeepSeek-R1 remind us that post-training can change model behavior dramatically.
9. **Convergence on the modern decoder recipe is real.** Combinations such as RoPE, RMSNorm, SwiGLU, and GQA are popular not because they are new, but because they are more realistic in both training and inference.
10. **The final question is always this: how will this architecture fail inside my system?** If you cannot answer that, model selection is not done.

For AI engineers, model architecture has never been abstract theory. It directly determines training cost, inference latency, context behavior, memory pressure, deployment form, and the way the system will eventually fail. **To understand architecture is, in essence, to understand what an AI system will look like in production.**

aiengineeringhandbook.com

14 References

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.

aiengineeringhandbook.com

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*.

aiengineeringhandbook.com

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.

aiengineeringhandbook.com