1. Foundations

Understanding the inputs, computation, and system cost of large language models through the lifecycle of a single model request.

Table of contents

1 Overview: The Lifecycle of a Model Request

> **When a user sends a message to an AI assistant, how does the system turn raw text into the next generated token, step by step?**

Make the scene more concrete. Suppose a user sends an internal AI assistant a message on a Monday morning:

> "Why did the payment service time out last night? Based on the logs and tickets below, give me a summary in no more than five sentences, and identify the most likely root cause."

For the user, this is just an ordinary question-answer exchange. For the system, it is a request that immediately triggers a full inference pipeline. The raw string is first split into tokens by the tokenizer, the long logs consume compute during the prefill stage, the accumulated history begins to occupy KV cache, the model emits logits at the final position, and the decoder then turns that probability distribution into the first visible token. Before the first character appears, latency, memory pressure, and cost have already been jointly determined by prompt length, model architecture, and system implementation.

This is the only question this chapter really needs to answer.
If you can explain this chain clearly, then tokenization, embeddings, attention, decoding, long context, and debugging stop being isolated concepts. They become different views of the same system.

This chapter follows one fixed pipeline from top to bottom:

text → tokens → embeddings → transformer → logits → decoding → text

This pipeline is both the smallest useful mental model for understanding LLMs and the most useful coordinate system for engineering debugging.
Many problems that look like "the model got worse" do not come from the weights themselves. They usually come from some point along this chain: tokenizer mismatch, context that has grown too long, KV cache expansion, decoding parameters drifting out of balance, or a serving system that turned the right distribution into the wrong behavior.

2 Input Representation

> **!** Important
>
> This section answers a few key questions first:
> 1. Why do modern LLMs use subword tokenization instead of word-level or character-level tokenization?
> 2. How does vocabulary size trade off against quality, sequence length, and compute cost?
> 3. Why can't the model use token IDs directly as numeric input?
> 4. Why is embedding lookup equivalent to multiplying a one-hot vector by an embedding matrix?
> 5. Why does a Transformer still need an explicit position representation?

The core conclusion of input representation is simple: **the model does not first see "text." It sees a sequence of encoded discrete symbols.**
For an LLM, the raw string must first be split into tokens, then mapped into vectors, and only then passed into the network. That is why input representation is not a preprocessing detail. It is part of the model interface.

2.1 Tokenization: Turning strings into symbol sequences

Tokenization maps raw text into a token sequence. It looks like preprocessing, but in practice it determines whether the model can stably cover domain terminology, multilingual text, numbers, code, and noisy input. It also directly determines sequence length and inference cost.

The reason mainstream LLMs do not use word-level tokenization is that word-level tokenization quickly runs into the OOV (out-of-vocabulary) problem. New words, morphological variation, spelling noise, and domain terms force the vocabulary to grow without bound. Character-level tokenization runs to the opposite extreme: it almost never suffers from OOV, but the sequence becomes much longer, and longer sequences make attention more expensive.

That is why modern LLMs broadly adopt **subword tokenization**. Its engineering value is not that it is somehow "more semantic," but that it balances two goals:

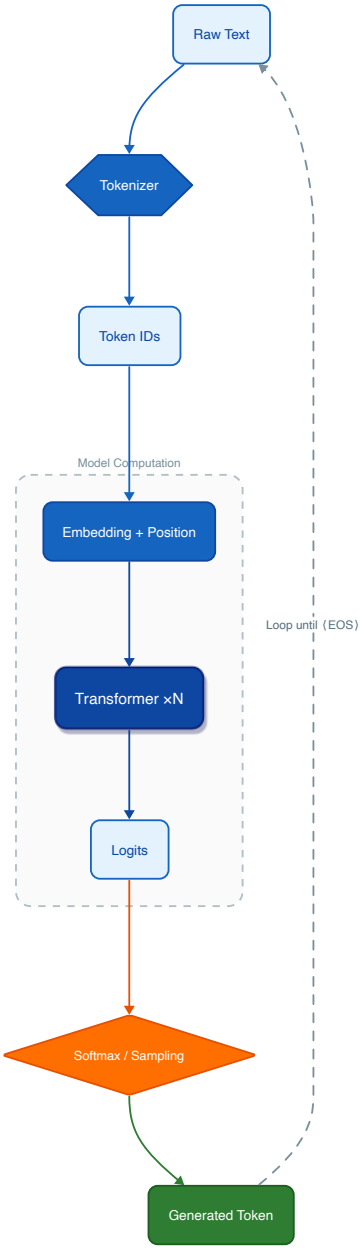

Figure 1: The lifecycle of a single LLM request: text → tokens → vectors → distribution → text

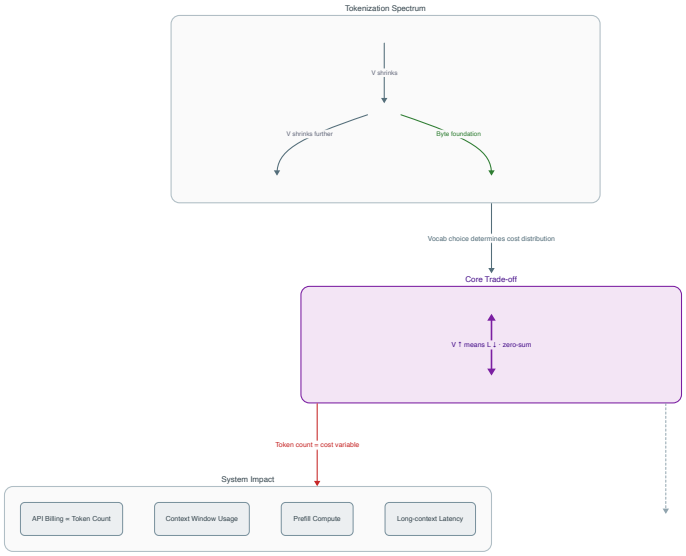

Figure 2: Input representation tradeoffs: vocabulary size, sequence length, and cost

- keep frequent patterns short, reducing token count;
- still represent rare patterns, avoiding OOV.

The three most common families are:

BPE (Byte-Pair Encoding)
Start from smaller units and repeatedly merge frequent adjacent fragments, gradually growing the vocabulary. It is simple, mature, and widely compatible.

WordPiece
Also performs merges, but leans more toward "which merges most improve language-model probability." You can treat it as a BPE-style variant with a language-model objective.

Unigram LM
Starts from a larger candidate vocabulary, then repeatedly removes the tokens that help overall likelihood the least. Given a string s, it chooses the segmentation with the highest probability among all valid segmentations:

$$\hat{t}_{1:m} = \arg \max_{t_{1:m}:\,\mathrm{concat}(t_{1:m})=s} \prod_{i=1}^{m} p(t_i)$$

The difference among these methods is, fundamentally, **how they learn symbol boundaries**.
If you work with multilingual corpora, code, or noisy data, byte-level BPE is often more robust because it first guarantees representability at the byte level, then keeps merging over bytes. It is not simply character-level tokenization. It is a more robust low-level encoding scheme.

A concrete example makes it easier to see how tokenization granularity rewrites system cost. Consider a common engineering string:

```
microservice_payment_v2 failed at 03:14
```

The splits below are only illustrative. They do not represent the actual output of a specific tokenizer, but they are enough to show the tradeoff:

Granularity	Illustrative split	Pressure on vocabulary size V	Pressure on sequence length L	Engineering implication
Word-level	`microservice_payment_v2` `failed` `at` `03:14`	Higher	Low	Identifiers and rare strings easily become OOV
Character-level	`m` `i` `c` `r` `o` …	Very low	Very high	Almost no OOV, but very long sequences

Granularity	Illustrative split	Pressure on vocabulary size V	Pressure on sequence length L	Engineering implication
Subword	`micro service _payment _v 2 failed at 03 : 14`	Medium	Medium	The mainstream compromise
Byte-level	Similar to subword, but unknown fragments can fall back to bytes	Medium	Medium to somewhat high	More robust for code, noisy text, and multilingual data

This example shows why the tokenizer is not a small detail.
If the same error log gets split into more tokens, token count goes up. Once token count goes up, API cost, context occupancy, and attention cost all rise together. Code identifiers, mixed numeric strings, URLs, file paths, and multilingual text are often exactly the inputs that drive fragmentation upward.

2.2 Vocabulary size, fragmentation rate, and cost

Tokenization does not only affect "how text is split." It also changes system cost.
The more finely a piece of text is split, the more tokens it contains. More tokens mean higher API cost, fuller context windows, and more expensive downstream attention. This is why many engineering teams care about **fragmentation rate**.

The tradeoff around vocabulary size can be understood with one simple relationship:

$$\text{embedding / softmax cost} \propto Vd, \qquad \text{attention cost} \propto L^2$$

where:

- V is vocabulary size;
- d is representation dimension;
- L is sequence length.

When the vocabulary gets larger, the embedding layer and output projection get heavier. When the vocabulary gets smaller, text gets split more finely, the sequence gets longer, and attention cost rises. There is no universally optimal vocabulary size. There is only a compromise point determined jointly by corpus distribution, language coverage, and deployment budget.

This is also why **you cannot directly compare perplexity across models that use different tokenizers**. Perplexity is defined over token sequences, but different tokenizers split the same text into different token boundaries and different token counts. Once the denominator changes, the numbers are no longer in the same coordinate system.

2.3 Embeddings: Mapping discrete symbols into vector space

After tokenization, what you have is token IDs. But an ID is only a label. It does not carry geometric meaning as a continuous number.
ID=100 is not "closer to a concept" than ID=3. If you feed IDs in directly as continuous values, the model will learn order relationships that do not actually exist.

The role of embeddings is to map discrete tokens into a continuous vector space. Let vocabulary size be V, embedding dimension be d, and the embedding matrix be

$$E \in \mathbb{R}^{V \times d}$$

For the t-th token, its embedding is

$$e_t = E[t]$$

If we write it as a one-hot vector $x_t \in \mathbb{R}^V$, the same operation can also be written as

$$e_t = x_t^\top E$$

This shows that embedding lookup looks like table lookup, but is really a **sparse linear map**.
That is why embedding is not just "looking up a vector." It is the interface layer between the world of discrete symbols and the world of neural computation.

Many decoder-only LLMs also use **weight tying**, which means the input embedding and the output projection share parameters:

$$W_{\text{out}} = E^\top$$

The reason is straightforward: reading in a token and predicting a token both operate in the same symbol space. Sharing parameters reduces the parameter count and often improves sample efficiency.

2.4 Position representations: Letting the model know order

Token embeddings alone are not enough, because attention does not carry a built-in sense of order.
If you give the model no positional information at all, "man bites dog" and "dog bites man" use almost the same set of tokens, and the order difference becomes hard to represent reliably.

The classic absolute positional encoding is sinusoidal positional encoding:

$$PE(t, 2i) = \sin\left(\frac{t}{10000^{2i/d}}\right), \qquad PE(t, 2i+1) = \cos\left(\frac{t}{10000^{2i/d}}\right)$$

It assigns a fixed vector to each position, then adds it to the token embedding. Why use multi-frequency sin / cos? Because a single frequency can only express displacement at one scale. A multi-frequency combination lets the model sense both short-range and long-range order patterns.

In modern LLMs, the more common choices are:

- **relative positional bias**: emphasize "distance" more than "index";
- **RoPE**: directly rotate Q/K as a function of position, so relative displacement appears in dot-product geometry;
- **ALiBi**: add a bias to attention scores that scales linearly with distance.

What matters here is not the method names, but the engineering implication: **the positional scheme affects the model's ability to extrapolate in length**.
If a model was trained only up to length 8K and you expect it to operate reliably at 128K, context failures are often not just "the model is not big enough." They may come from the position representation beginning to distort.

2.5 Summary

The three things most worth remembering about the input-representation layer are:

1. the tokenizer determines what symbols the model sees, and also determines token count and cost;
2. embeddings map discrete symbols into a learnable vector space;
3. positional information determines whether the model can reliably understand order and length.

From an engineering perspective, token count is itself a cost variable.
If the same user input gets split into more tokens, it simultaneously raises:

- API billing;
- context occupancy;
- prefill compute;

- end-to-end latency under long context.

That is why input representation is never just "prep work before the model." It is the first layer of system cost.

3 Transformer Blocks

> **!** Important
>
> This section answers a few key questions first:
> 1. How does scaled dot-product attention work, and why do we divide by $\sqrt{d_k}$?
> 2. What is a causal mask, and why must a decoder-only model have it?
> 3. Why is multi-head attention not something a "single big head" can replace?
> 4. If attention already mixes information across tokens, why do we still need an FFN/MLP?
> 5. Why are residual connections and normalization so critical for deep Transformers?

The essence of a Transformer is not "a lot of linear algebra." It is a specific division of computational labor.
In a typical decoder block, the system first routes information across tokens through attention, then performs nonlinear transformation within each individual token through the FFN, and finally relies on residual connections and normalization to keep the deep network trainable.

In pre-norm form, a basic block can be written as:

$$h^{(l+\frac{1}{2})} = h^{(l)} + \text{Attention}(\text{Norm}(h^{(l)}))$$

$$h^{(l+1)} = h^{(l+\frac{1}{2})} + \text{FFN}(\text{Norm}(h^{(l+\frac{1}{2})}))$$

These two lines already capture the main computational pattern of modern LLMs.

3.1 Attention: A content-based retrieval system

Given input hidden states $X \in \mathbb{R}^{L\times d}$, the model first projects them into three sets of vectors:

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

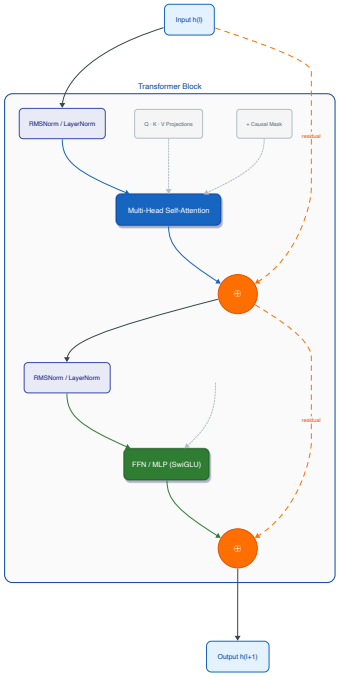

Figure 3: The internal structure of a single Transformer block (pre-norm)

where:

- **Q (query)**: what information I am looking for;
- **K (key)**: what matching signal I can offer;
- **V (value)**: what content I actually contribute if I am attended to.

Standard scaled dot-product attention is written as:

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

where M is the mask matrix.
The reason for dividing by $\sqrt{d_k}$ is to avoid dot-product variance becoming too large as dimension increases, which would cause softmax to saturate too early.

The most stable intuition is this: **attention is fundamentally a content-based retrieval system**.
Each position takes its query, matches it against the keys across the full sequence, and decides which values to read from. It does not read from a fixed window. It routes information dynamically based on current content.

For example, in the sentence "After the server restarted, it resumed service," the token "it" is more likely to read from the representation of "server" than from the action "restarted." Attention is exactly this kind of context-dependent selective reading.

3.2 Causal masking and multi-head attention

For a decoder-only model, the autoregressive objective requires that position i must not see future positions $j > i$ when predicting the next token.
That is what the causal mask does: it sets future-position scores to extremely small values, so their probabilities become 0 after softmax. Without the causal mask, the model would peek at the answer during training, and the training objective would no longer match the inference process.

The point of multi-head attention (MHA) is **division of labor**.
If you use only one large head, the model must use the same representation to learn local syntax, long-range dependencies, format boundaries, and other relational patterns at once. Multiple heads let the system learn different attention patterns in parallel in different subspaces, which improves both expressiveness and stability.

When the query comes from one sequence while the key/value come from another, you get **cross-attention**. This is very common in encoder–decoder models and multimodal models.

3.3 FFN: Creating features within a single position

Attention decides "where to read," but it does not decide "how to process what was read."
That is the role of the FFN (Feed-Forward Network). Its most common form is:

$$\mathrm{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2$$

Without this nonlinear sublayer, stacked linear transforms could collapse into a single linear transform, and model expressiveness would be sharply limited.
So attention and FFN are not substitutes. They are a division of labor:

- attention: information exchange across tokens;
- FFN: feature transformation within a single token.

In modern LLMs, the FFN is often one of the dominant contributors to both parameter count and compute. Many models first expand the dimensionality, then project it back down; gated SwiGLU structures further improve representational efficiency. You can think of it this way: attention routes information, FFN manufactures features.

3.4 Residual connections and normalization: Why deep networks still train

The form of a residual connection is simple:

$$x_{l+1} = x_l + f(x_l)$$

But its importance is large. It gives the network an explicit identity path, so each layer does not have to start over. It can learn an incremental correction on top of the old representation. Gradients also propagate more easily through deep networks.

Normalization controls representation scale. In Transformers, the most common choices are:

- **LayerNorm**: normalizes mean and variance;
- **RMSNorm**: scales only by the root mean square, making it simpler and a better fit for many modern decoder-only recipes.

Another key choice is **pre-norm** versus **post-norm**. Most modern deep LLMs prefer pre-norm because it is usually more stable in large-scale training.

3.5 Summary

The single sentence most worth remembering about a Transformer block is:

attention handles information routing across tokens, FFN handles nonlinear transformation within a token, and residual connections plus normalization keep the deep network trainable.

Seen from the perspective of older neural architectures, the biggest difference from an RNN is this:
During prefill, a Transformer can process the whole sequence in parallel. It does not depend on recursive hidden state being passed along time step by time step. It uses attention for learnable information selection instead of relying on a fixed convolution window or unidirectional recurrent state propagation. That is both why it is so expressive and why it gets expensive under long context.

4 From Hidden States to Probabilities

> **!** Important
>
> This section answers a few key questions first:
>
> 1. What is the relationship among hidden states, logits, and probabilities?
> 2. Why does the model output logits instead of directly outputting a token?
> 3. What is $P(x_t \mid x_{<t})$, and how does it relate to the training objective?
> 4. Why can the same model weights stay fixed while runtime behavior changes a lot?

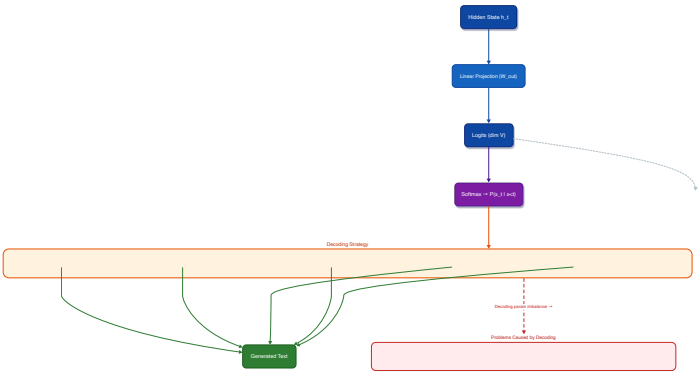

Figure 4: From hidden states to logits, then to decoding strategies

After passing through many Transformer layers, what the model actually produces is a sequence of **hidden states**.
For autoregressive generation, the most important one is the hidden state at

the final position, h_t, because it is the internal representation of "what the next token should be under the current context."

4.1 hidden state → logits

The model does not interpret a hidden state directly as a word. It first applies a linear projection that maps it back into vocabulary space:

$$z = h_t W_{\text{out}} + b$$

Here z is the **logits**.
Its dimensionality equals the vocabulary size V, and each component is the unnormalized score of one candidate token. Logits are neither probabilities nor final output text. They are simply the raw scores the system assigns to "how appropriate each token is."

4.2 logits → probabilities

Once you have logits, the system uses softmax to turn them into a probability distribution:

$$p_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

This gives you the probability of each candidate token under the current context. From a modeling perspective, what a language model learns is a sequence of conditional probabilities:

$$P(x_t \mid x_{<t})$$

In other words, the model is not "writing an answer." It is repeatedly estimating "what the next token is most likely to be."

4.3 Why output logits first instead of directly outputting a token

This is an important question that is often overlooked.
The model outputs logits first instead of directly outputting a token for three main reasons:

First, **training needs a differentiable objective**.
If the model made a discrete choice directly, stable gradient propagation would be much harder. Logits plus softmax provide a continuous interface that can be optimized.

Second, **runtime needs to preserve a space of choices**.
If the model immediately gave you only one token, you could no longer adjust temperature, apply top-k/top-p sampling, add repetition penalties, or perform constrained decoding at decode time. Logits allow "the model distribution" and "the runtime policy" to be decoupled.

Third, **system behavior is not determined by weights alone**.
The weights define a distribution. Runtime decides how to select from that distribution. This also explains why changing only the decoding parameters can make the same model look like it has "a different personality."

4.4 The training objective: Why next-token prediction is central

During training, language models typically minimize the negative log-likelihood of the true next token:

$$\mathcal{L} = -\sum_{t} \log P(x_t \mid x_{<t})$$

This is next-token prediction.
Its significance is not that "the model only guesses the next word." It is that by repeatedly optimizing local conditional probabilities, the model eventually learns statistical structure across tokens, across sentences, and across documents.

One engineering point is especially important:
training learns a distribution; inference exposes a policy.
If you mix up those two things, you will misdiagnose problems that really belong to decoding or the serving system as "model capability problems."

5 Decoding

> **!** Important
>
> This section answers a few key questions first:
>
> 1. What exactly do greedy decoding, temperature sampling, top-k, and top-p do?
> 2. What does temperature change mathematically, and what problems do very low and very high temperature create?
> 3. Why does beam search often make open-ended generation dull?
> 4. Why can the same model become repetitive, hallucinatory, or unstable when only the decoding parameters change?
> 5. Why do structured output and tool calling depend so heavily on stop design and constrained decoding?

The model only provides a probability distribution. **Decoding** turns that distribution into actual text.
That is why decoding is not "a small trick at the last step." It is a direct control lever on system behavior.

5.1 Greedy, temperature, top-k, top-p

The simplest method is **greedy decoding**:
At each step, choose the token with the highest probability. It is fast and deterministic, but it is also easy for it to become templated or fall into repetition.

If you want to preserve diversity, you first rescale the logits with temperature:

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

where:

- $T < 1$: the distribution becomes sharper, more conservative, and closer to deterministic;
- $T > 1$: the distribution becomes flatter, more random, and also more prone to distortion.

Then the common sampling strategies are:

top-k
Keep only the top k highest-probability tokens, then sample from them. The advantage is simplicity. The downside is that the candidate boundary is fixed, so it is not always stable across prompts.

top-p (nucleus sampling)
Do not fix the number of candidates. Instead, take the smallest candidate set whose cumulative probability reaches p. It is more adaptive, so it is usually more stable for open-ended generation.

beam search
Keep several high-scoring paths in parallel, favoring high-likelihood output. But precisely because it prefers high likelihood, it often feels conservative and homogeneous in open-ended dialogue or creative tasks.

5.2 Why decoding creates problems

Many "model problems" are really decoding problems.

Hallucination
If the model itself is uncertain about the answer, but you use a relatively high temperature and a large top-p without grounding or citation constraints, low-quality tokens from the tail of the distribution become easier to sample. The result is high-confidence hallucination.

Repetition
If the model enters a high-probability self-loop and you use greedy decoding or extremely low temperature, it may keep sliding down the same locally high-scoring path. Repetition penalties can help, but penalties that are too strong may also damage legitimate repetition in code, JSON, tables, and fixed templates.

Instability
In structured-output scenarios, a slightly higher temperature, a slightly weaker stop sequence, or a slightly fuzzier JSON boundary can suddenly turn output from "fully parseable" into "almost unusable." That is not a small quality fluctuation. It is a product-level failure.

5.3 Engineering tradeoffs: quality, randomness, and stability

From an engineering perspective, decoding is a tradeoff among three goals:

- **quality**: is the answer accurate and useful;
- **randomness**: is the output diverse enough not to feel mechanical;
- **stability**: is the format, structure, and tool call behavior reliable.

That is why different tasks should use different strategies:

- **tool calling, JSON, structured output**: low temperature, and constrained decoding when needed;
- **chat Q&A**: moderate temperature and top-p, balancing naturalness and stability;
- **brainstorming, candidate generation**: higher randomness can be useful, but it must be followed by a ranker or verifier.

The stop sequence is especially critical here.
For tool calling, code completion, JSON output, and similar scenarios, stop is not "just an ending marker." It is a boundary-control mechanism. If stop is poorly designed, the model will over-generate, truncate at the wrong place, or glue structured output to ordinary text.

One-line summary of this section: **weights determine the probability distribution; decoding determines system behavior.**

6 Architecture Variants

> **!** Important
>
> This section answers a few key questions first:
>
> 1. What kinds of tasks are encoder-only, decoder-only, and encoder–decoder architectures each suited for?
> 2. Why are most modern chat LLMs decoder-only?
> 3. In what scenarios would you choose a BERT-style model over a GPT-style model?
> 4. What role does MoE play here, and why is it not "another task architecture"?
> 5. Which class of architectural choice is most likely to dominate inference memory during decode?

"Architecture" does not just answer "how many layers does the model have." It answers **how the model reads context, how computation is organized, and which side of the system bears the cost**.

In engineering, architecture selection usually starts with five questions, not with memorizing three names:

1. Is the task representation learning, or open-ended generation?
2. Do input and output naturally split into two stages?
3. At serving time, is prefill more expensive, or decode?
4. Are you willing to accept a more complex inference path in exchange for stronger input modeling?
5. Are you trying to increase total parameter count, or to reduce per-step activation cost?

You can first grab the main thread with the decision table below:

Primary need	More natural architecture choice	Why	Typical cost
Classification, retrieval, ranking, reranking	encoder-only	Bidirectional modeling is better suited to representation learning	Not suitable for long-form autoregressive generation
Chat, completion, code generation, tool use	decoder-only	Training objective and inference interface are unified	Decode path and KV cache cost stand out

Primary need	More natural architecture choice	Why	Typical cost
Translation, summarization, rewriting, strict seq2seq	encoder–decoder	Input and output are clearly separated	The system path is more complex and heavier to serve
Expanding total parameters under fixed FLOPs	MoE (stacked onto a main architecture)	Sparse activation improves parameter efficiency	Routing, load balancing, and communication become more complex

In other words, architecture first determines **the product interface, the inference path, and the cost curve**, and only secondarily the category names used in papers.

6.1 Encoder-only

Encoder-only models use bidirectional attention, so each position can see both the left and right context.
They are very good at representation learning, which is why they are commonly used for classification, matching, ranking, retrieval, and reranking. BERT, RoBERTa, and DeBERTa all belong to this family.

If your task is fundamentally "encode the input into a high-quality representation," rather than "continue generating a long output," encoder-only is often the more natural choice.

6.2 Decoder-only

Decoder-only models use a causal mask and can only see the content before the current position.
Its training objective and inference procedure line up naturally:

$$P(x_t \mid x_{<t})$$

This is exactly the dominant paradigm for today's chat models. GPT, LLaMA, Mistral, Qwen, and similar systems all belong to this class.

Why do modern chat LLMs almost all choose decoder-only? Because it unifies three things into one interface:

- the training objective is next-token prediction;
- the inference process is token-by-token autoregressive generation;

- product input can be uniformly expressed as "continue completing the current context."

This makes the system interface extremely simple: system prompts, user messages, retrieved passages, and tool returns can all be concatenated into one context, then the model is asked to keep writing.

6.3 Encoder–decoder

An encoder–decoder architecture separates input processing from output generation. The encoder first performs bidirectional encoding of the input, and the decoder then generates autoregressively conditioned on the encoder output.
It is naturally well-suited to "input sequence → output sequence" tasks such as translation, summarization, and rewriting. T5 and BART are canonical examples.

Compared with decoder-only, its advantage is more complete modeling of the input. Its cost is a more complex architecture and inference path.

6.4 PrefixLM and MoE: Two common supplemental concepts

PrefixLM
You can think of it as a mixed masking scheme: the prefix can be read bidirectionally, while the suffix is generated causally. What it demonstrates is that **the mask itself is part of task design**.

MoE (Mixture of Experts)
MoE is not a task architecture like encoder or decoder. It is a way to scale parameter count. It usually appears in the FFN: a token first goes through a router, then activates only a small number of experts.

$$\mathrm{MoE}(x) = \sum_{k \in \mathrm{TopK}(g(x))} g_k(x)\, E_k(x)$$

The benefit of MoE is that total parameter count can be very large while only a small subset is activated at each step.
Its costs are equally clear: routing quality, load balancing, and distributed communication can all become new bottlenecks.

6.5 Summary

If you look only at the inference system, one key fact about decoder-only models is this:
during decode, inference memory is usually not dominated by parameters first. It is dominated by KV cache first.
That is, what makes a 70B model difficult to serve under long context and high

concurrency is often not that "the weights are too large," but the cost of caching historical context.

That is also why many modern decoder recipes converge on a similar combination: RoPE, RMSNorm, SwiGLU, and GQA. They are not accidentally popular. They form a strong engineering equilibrium among quality, training stability, and inference cost.

7 Long Context Is a Systems Problem

! Important

This section answers a few key questions first:

1. Why does long context create quadratic attention cost?
2. Why do modern inference systems split prefill and decode into two distinct stages?
3. Why does KV cache exist, and how does its memory cost grow?
4. When the context window expands from 4K to 128K, what gets expensive first, and what breaks first?
5. Why is long context both a modeling problem and a systems problem?

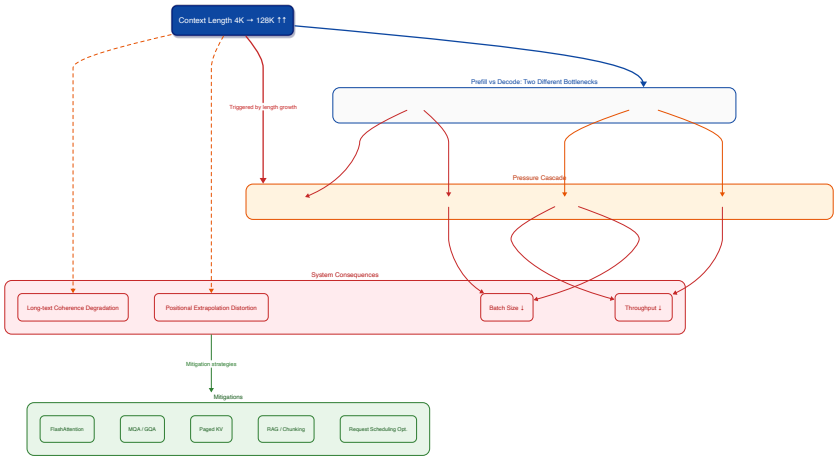

Figure 5: Why long context turns from a modeling problem into a systems problem

Long context is most often misunderstood as "just increase the window size."

But for a real system, it is both a modeling problem and a systems problem: the model has to understand longer dependencies, while the system has to bear higher compute, memory, and bandwidth cost.

7.1 Prefill and decode: Two different physical stages

An autoregressive inference run is usually divided into two stages:

Prefill
Process the existing context. The model must run a full forward pass over the entire prompt. Compute is high, and the workload is typically more compute-bound.

Decode
Generate new content token by token. Each step adds only one new position, but the system must repeatedly read historical cache, so it is more likely to be constrained by memory and bandwidth.

That is why many modern inference systems explicitly distinguish prefill from decode.
They have different bottlenecks, and therefore different optimization strategies. Prefill cares more about attention kernels, batching, and context length. Decode cares more about KV cache, memory layout, and bandwidth efficiency.

7.2 Quadratic attention and linearly growing cache

The core cost of standard attention during prefill comes from the $L \times L$ score matrix, so when sequence length doubles, both compute and memory pressure rise sharply. That is why long prompts often drive up time to first token (TTFT).

The key cost during decode, by contrast, comes from **KV cache**.
To avoid recomputing the entire historical context every time a new token is generated, the system caches the key/value for all past positions. The approximate cost per layer can be written as:

$$\text{KV bytes} = B \times L \times H_{kv} \times d_h \times 2 \times s$$

where:

- B: batch size
- L: sequence length
- H_{kv}: number of KV heads
- d_h: dimension per head
- 2: one copy each for K and V
- s: bytes per numeric value

If the model has N layers, total cost must be multiplied by the number of layers N.

This formula is extremely important because it explains why the decode stage is often constrained by cache before it is constrained by parameters.

A quick order-of-magnitude example makes this more concrete. Suppose a decoder-only model has 80 layers, $H_{kv} = 8$, $d_h = 128$, uses FP16/BF16 ($s = 2$ bytes), serves a single request with batch size $B = 1$, and has context length $L = 32768$. Then per-layer KV cache is roughly:

$$32768 \times 8 \times 128 \times 2 \times 2 \approx 128 \text{ MB}$$

Multiply by 80 layers, and total KV cache for the model is about 10 GB. If batch size rises to 4, KV cache alone approaches 40 GB, before accounting for model weights, activations, allocator fragmentation, and other runtime overhead.

This is why, once long context meets high-concurrency serving, the problem immediately shifts from "can the model read it?" to "can the system fit it, and can it still run fast?" GQA, MQA, paged KV management, and more aggressive request scheduling all exist because of this constraint.

7.3 Why long context hurts both latency and throughput

When you expand the window from 4K to 128K, one thing does not happen. Several pressures stack on top of each other:

- prefill compute rises sharply;
- KV cache grows linearly;
- batch size is often forced down by memory constraints;
- bandwidth pressure during decode increases;
- positional extrapolation, retrieval precision, and long-document consistency may also degrade.

So long context is not only "can the model ingest it." It is also "can the system still serve it efficiently."

7.4 Common mitigation strategies

Common engineering mitigations include:

- **KV cache**: avoid recomputing history during decode;
- **MQA / GQA**: directly shrink cache by sharing or group-sharing K/V;
- **FlashAttention**: optimize implementation to reduce HBM traffic;
- **better request scheduling and memory management**: for example, paged KV management;
- **RAG / chunking / summaries**: do not force all information into context; move part of the problem into the system layer.

That is also why long-context capability often should not be understood as "the larger the window, the better."
For many business problems, organizing information through retrieval, chunking, summaries, and structured state is more reliable and cheaper than brute-forcing a giant context window.

One-line summary of this section: **long context is not a single model capability. It is the result of attention complexity, cache management, positional extrapolation, and system scheduling acting together.**

8 AI Engineer's Debugging Handbook

> **!** Important
>
> This section answers a few key questions first:
>
> 1. If model output suddenly becomes chaotic, which layer should you check first?
> 2. If latency suddenly rises, should you look at prefill first or decode first?
> 3. If memory usage grows abnormally, why should you suspect KV cache first?
> 4. Which failures look like "model capability problems" but are really tokenizer, template, or decoding problems?

The real value of fundamentals is not how many formulas you can recite in an interview. It is knowing where to look first when something breaks. The table below can serve as a minimal debugging notebook for an AI engineer.

Symptom	First checkpoint	Common root cause	Next action
Garbled output, role confusion, broken tool argument boundaries	tokenizer / special tokens / chat template	training and inference are using different protocols	align tokenizer, special tokens, and template
First token is very slow	prefill / prompt length / attention kernel	context is too long, and prefill is dominated by attention cost	shorten context, apply chunking, inspect kernel and batching

Symptom	First checkpoint	Common root cause	Next action
Generation gets slower over time, memory keeps rising	KV cache / context length / batch	cache grows during decode, bandwidth becomes limiting	inspect context length, KV management, and whether GQA/MQA is enabled
Output repeats, becomes templated, or falls into loops	decoding	greedy or low-temperature self-loops; repetition penalty is mis-set	adjust temperature, top-p, repetition penalty
JSON / tool call behavior is unstable	stop sequence / constrained decoding / template	boundaries are unclear, structural constraints are too weak	add stop, and use constrained decoding when necessary
Long-document QA becomes contradictory	retrieval / chunking / positional scheme	key evidence never entered context, or length extrapolation failed	debug retrieval and chunking first, then inspect positional scheme and window strategy
Poor understanding of domain terminology or code identifiers	tokenizer / training data	excessive fragmentation or insufficient domain coverage	inspect fragmentation rate, term normalization, and continued pretraining
Same model is fine offline but worse online	serving pipeline	preprocessing, template, stop, or decoding settings do not match	align online and offline configuration, run end-to-end replay

The most important debugging principle here is:

do not blame the model weights first.
First walk the chain: is input representation correct? Is context organized correctly? Where is the stage bottleneck in the inference system? Did decoding turn a healthy distribution into pathological behavior?

If you have this pipeline as a mental model, many problems become easier to localize.

9 Chapter Summary

This chapter re-explained how large language models work by following a minimal but complete inference pipeline.

- text is first split into tokens by the tokenizer;
- tokens are mapped into embedding space and augmented with positional information;
- multiple Transformer blocks repeatedly rewrite the representation through attention and FFN;
- the hidden state at the last position is projected into logits;
- softmax turns logits into a probability distribution;
- decoding turns that distribution into actual text;
- long context and the inference system determine whether this chain can run in the real world at an acceptable cost.

If you remember only one sentence, make it this:

An LLM is not "a program that can talk." It is an engineering chain that maps symbols to distributions, then distributions to behavior.

9.1 Question Recap

1. What is a token, and why do modern LLMs prefer subword tokenization?
A token is the basic discrete symbol the model uses to process text. Subword tokenization gives the most practical compromise between word-level OOV and character-level long sequences.

2. Why can't token IDs be used directly as input?
Because IDs are only labels. They do not carry geometric meaning in continuous space. Embeddings map discrete labels into a learnable vector space.

3. Why does a Transformer need positional information?
Because attention has no built-in sense of order. Without positional representation, the model cannot reliably distinguish word order.

4. What does the model output before it actually generates text?
It first outputs hidden states, then applies a linear projection to get logits, and only after softmax does it obtain a probability distribution.

5. Why does a language model generate a probability distribution instead of directly outputting a word?
Because training needs a differentiable objective, and inference needs to preserve runtime choice space. Logits let the model distribution and decoding policy be decoupled.

6. Why does scaled dot-product attention divide by $\sqrt{d_k}$?
To control the scale of dot-product scores and prevent softmax from saturating too early as dimension grows.

7. Why is the causal mask necessary?
It prevents the model from peeking at future tokens and keeps training aligned with autoregressive inference.

8. Why is attention $O(L^2)$?
Because every position interacts with every other position, producing an $L \times L$ score matrix.

9. What is the difference between logits and probabilities?
Logits are unnormalized scores. Probabilities are the normalized distribution after softmax.

10. What do temperature, top-k, and top-p each change?
Temperature changes the sharpness of the distribution. Top-k restricts sampling to a fixed number of candidates. Top-p restricts it to a cumulative-probability candidate set. Together they control the balance between randomness and stability.

11. Why does long context dramatically increase inference cost?
Because prefill attention cost rises sharply with length, while decode-side KV cache also grows linearly and puts pressure on memory and bandwidth.

12. What happens inside the inference pipeline after a prompt arrives?
Text is first tokenized, then mapped into embeddings; the model computes hidden states through multiple Transformer layers; finally it emits logits, which are decoded into new tokens, and the cycle repeats.

13. Why does the system distinguish prefill and decode?
Because the two stages have different bottlenecks: prefill is more compute-bound, while decode is more constrained by cache and bandwidth.

14. Why does KV cache exist?
To avoid recomputing the key/value of the full historical context at every autoregressive step. It trades memory for decode efficiency.

15. Why is decoding strategy a system-design problem?
Because it directly affects output quality, randomness, format stability, and inference cost, not just "writing style."

16. Where should you look first when output becomes chaotic, latency spikes, or memory explodes?
Start from the pipeline: tokenizer and templates, context length and prefill, KV cache and decode, decoding parameters and stop design.

10 References

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

2. Sennrich, Haddow, Birch. *Neural Machine Translation of Rare Words with Subword Units*. 2016.

3. Kudo, Richardson. *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*. 2018.

4. Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013.

5. Pennington, Socher, Manning. *GloVe: Global Vectors for Word Representation*. 2014.

6. Press, Wolf. *Using the Output Embedding to Improve Language Models*. 2017.

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

8. Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2021.

9. Press et al. *Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation*. 2022.

10. Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022.

11. Ainslie et al. *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. 2023.

12. Gu, Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. 2023.

13. Holtzman et al. *The Curious Case of Neural Text Degeneration*. 2020.

14. Ba, Kiros, Hinton. *Layer Normalization*. 2016.

15. Zhang, Sennrich. *Root Mean Square Layer Normalization*. 2019.