Lecture 1: Overview & Tokenization

Language Modeling from Scratch · lecture notes · rebuilt for NVIDIA Blackwell, late 2026

Section 1: Leaky Abstractions, Scale Transfer, and the Efficiency Lens

Modern language modeling has undergone a rapid shift in how researchers and engineers interact with models. Visualizing this history as a three-tier vertical stack makes the trajectory unmistakable. At the bottom tier, in **2016**, building a language model meant **training your own model** from scratch with full control over the training data, model architecture, and training loop. An arrow pointing upward leads to the middle tier in **2018**, when pretrained checkpoints like BERT changed the workflow to **download and fine-tune**. A second arrow leads to the top tier **today**, where most practitioners simply **prompt API models** through hosted GPT, Claude, or Gemini endpoints. Beneath the entire three-tier stack sits the premise of building language models from scratch: **tearing up the whole stack** to regain visibility and control from raw bytes all the way to post-training.

Climbing up layers of abstraction boosts short-term productivity, which is why software engineering relies heavily on mature abstractions such as programming languages and operating system kernels. In language modeling, however, researchers are growing disconnected from the underlying technology precisely when the boundaries between layers are still unsettled. Fundamental research still requires modifying tokenization, model architectures, execution kernels, training data, and reinforcement learning together.

> **DEFINITION**
> **Leaky abstraction:** An abstraction layer whose internal implementation details cannot be safely ignored by its users because failure modes, performance bottlenecks, or design changes in lower layers bleed directly into higher-level behavior.

Because language model abstractions are leaky, treating a model purely as an opaque API endpoint prevents you from diagnosing failure modes or pushing the frontier. The guiding philosophy of this material is **understanding via building**: full-stack literacy is necessary to understand why models behave as they do and how to improve them.

At the same time, building language models from scratch faces an immediate practical obstacle: the industrialization of frontier models. Frontier training runs are far beyond academic budgets. In 2023, GPT-4 reportedly cost $100M to train, and in 2025, xAI built a 230K-GPU cluster to train Grok. Alongside the rising capital cost, closed technical reports routinely withhold architecture details, data mixtures, and training recipes.

COMMON MISCONCEPTION

You might think that studying small models (under 1B parameters) is a direct miniature replica of frontier training—just a scaled-down version that behaves identically in every respect. That assumption breaks for two concrete reasons:

1. **Compute mix shifts:** The fraction of floating-point operations (FLOPs) spent in attention versus multi-layer perceptrons (MLPs) changes substantially as model width and sequence length vary.
2. **Emergent behaviors:** Certain capabilities remain near zero at small parameter counts and then rise sharply only at large scale.

This creates the central methodological challenge: how do we study frontier language modeling rigorously when we cannot execute frontier-scale training runs ourselves? Answering that requires separating what transfers reliably across scales from what does not.

Knowledge Type	What It Covers	Transfers to Frontier?
Mechanics	How components work: Byte-Pair Encoding (BPE), Transformers, attention, kernels, parallelism, scaling laws, reinforcement learning (RL)	**Yes**
Mindset	Squeezing the most out of hardware; taking resource accounting and scaling laws seriously	**Yes**
Intuitions	Which specific data mixes or architectural tweaks maximize accuracy	**Partially**

Many empirical modeling choices lack clean theoretical justification. For example, Shazeer (2020) introduced the SwiGLU activation function based on experimental gains rather than a first-principles mathematical proof. Because small-scale empirical intuitions can shift at larger scale, we prioritize transferable mechanics and a disciplined resource-accounting mindset while treating small-scale empirical intuitions with caution.

That mindset is captured by the **Bitter Lesson** and the **efficiency lens**. A common misreading of the Bitter Lesson claims that “only scale matters; algorithms do not.”

COMMON MISCONCEPTION

You might think the Bitter Lesson implies that algorithmic and systems engineering are futile because throwing more compute at a problem always wins. That breaks because resource budgets are always finite, and any algorithmic waste multiplies directly with the size of the training cluster. The right reading of the Bitter Lesson is that **algorithms that scale are what matter**.

We can express the relationship between capability, algorithmic quality, and raw scale in a single conceptual equation:

$$\text{accuracy} = \text{efficiency} \times \text{resources}$$

Break this equation into its three terms:

- resources represents the total hardware and data budget you bring to a training run—compute (FLOPs), memory capacity, memory bandwidth, and training tokens.
- efficiency represents how effectively your algorithms, architecture, data pipeline, and systems software convert each unit of those resources into model capability.
- accuracy represents the resulting model capability achieved under that budget.

Taken as a single conceptual unit, the equation states that capability is a product of what you spend and how well you spend it. Efficiency therefore matters *more*, not less, at larger scale: doubling efficiency on a small run saves modest compute, whereas doubling efficiency on a frontier cluster multiplies the effective scale of the entire run. History bears this out: between 2012 and 2019, algorithmic advances delivered a $44\times$ efficiency gain on ImageNet alone. This yields our central engineering framing: **what is the best model we can build for a fixed compute and data budget?**

WORKED EXAMPLE

Consider two consequences of this framing.

First, quantify the efficiency multiplier using the historical $44\times$ ImageNet benchmark. Suppose a baseline training recipe requires 10^{23} FLOPs to reach a target accuracy. Applying algorithmic improvements that raise efficiency by a factor of 44 lets you reach the same effective product $\text{efficiency} \times \text{resources}$ using only:

$$\frac{10^{23}\ \text{FLOPs}}{44} \approx 2.27 \times 10^{21}\ \text{FLOPs}$$

Conversely, holding resources fixed at 10^{23} FLOPs with that $44\times$ efficiency gain matches the capability of a baseline run consuming 4.4×10^{24} FLOPs.

Second, verify why the compute mix shifts across scales. Recall from linear algebra that multiplying an $L \times d$ matrix by a $d \times L$ matrix costs on the order of $L^2 d$ operations (the sequence-length scaling of attention scores), while projecting an $L \times d$ activation matrix through a $d \times d$ weight matrix scales as Ld^2 (the width scaling of an MLP). Taking the ratio of attention scaling to MLP scaling gives:

$$\frac{L^2 d}{L d^2} = \frac{L}{d}$$

At a small model width of $d = 512$ and a sequence length of $L = 8{,}192$, this ratio is $8{,}192/512 = 16$, so attention dominates. At a wider model with $d = 6{,}144$ and a shorter sequence length of $L = 1{,}024$, the ratio flips to $1{,}024/6{,}144 \approx 0.17$, so the MLP dominates. An

> engineering optimization that speeds up only attention behaves very differently in these two regimes.

Maximizing capability under fixed resources unifies every layer of the language modeling stack. Picture a hub-and-spoke diagram with **Fixed Resources** (**Compute · Memory · BW · Data**) at the center, radiating outward with arrows to five surrounding engineering components:

1. **Tokenization:** Raw bytes are elegant and universal, but they are compute-inefficient when paired with quadratic attention.
2. **Architecture:** Key-Value (KV) sharing, sparse attention, and Mixture of Experts (MoE) cut memory and FLOPs per token.
3. **Systems:** Custom kernels, multi-GPU parallelism, and inference serving minimize data movement.
4. **Scaling laws:** Predictable scaling recipes allow us to tune hyperparameters cheaply on small models before committing to large runs.
5. **Data filtering:** Filtering and deduplication prevent burning FLOPs on low-quality or duplicate tokens.

Beneath this hub-and-spoke diagram lies a strategic transition: **today, frontier training is compute-constrained; tomorrow, as compute continues growing faster than high-quality human text, it becomes data-constrained.** We will need this fixed-resource lens throughout every technical unit that follows, starting with how real frontier models have evolved under these exact constraints.

KEY IDEA

Language model abstractions are leaky, so genuine mastery requires understanding the full stack from data and tokenization through kernels and alignment. Because $\text{accuracy} = \text{efficiency} \times \text{resources}$, maximizing capability under a fixed budget unifies tokenization, architecture, systems, scaling laws, and data curation—and while small-scale empirical intuitions transfer only partially, mechanics and an efficiency mindset transfer directly to frontier scale.

✎ Check Your Understanding

1. **Yes or No:** Does a small model (under 1B parameters) always accurately reflect the performance bottlenecks and capabilities of a frontier model? Justify your answer in one to two sentences.
   Answer: No. Small models experience different compute mixes (the ratio of attention to MLP FLOPs shifts with width and sequence length) and miss emergent behaviors that remain near zero until large scale.
2. **What would happen if** an engineering team discovers a $2\times$ algorithmic efficiency improvement while keeping their fixed hardware compute budget unchanged?
   Answer: Under $\text{accuracy} = \text{efficiency} \times \text{resources}$, doubling efficiency at fixed resources yields the same effective capability as doubling the compute cluster without buying additional hardware.

Section 2: Evolution of the Language Model Landscape and Late-2026 Frontier Systems

Before examining individual components in detail, we need to understand how modern language models arrived at their current form, what distinguishes historical baselines from late-2026 frontier systems, and how the operational requirements placed on language models have changed.

The technical lineage of language modeling unfolds along a three-stage historical timeline running from left to right across a horizontal axis:

1. **Pre-Neural ($<$2010s):** Starting with Claude Shannon (1950), who measured the entropy of written English, statistical language modeling relied on **n-gram models** (scaled massively by 2007), which powered classic machine translation (MT) and automatic speech recognition (ASR).
2. **Neural Ingredients (1997–2019):** Over two decades, researchers assembled the mathematical and systems building blocks of modern neural models: Long Short-Term Memory networks (LSTM, 1997), neural language models (Bengio, 2003), sequence-to-sequence models and the Adam optimizer (2014), attention mechanisms (2015), the Transformer architecture and Mixture of Experts (MoE, 2017), and distributed model parallelism frameworks including GPipe, ZeRO, and Megatron (2018–2019).
3. **Early Pretraining (2018–2019):** Combining these ingredients established the pretrain-and-fine-tune transfer learning paradigm through ELMo (LSTM-based), BERT (Transformer-based), and T5 (11B parameters, unifying tasks into a text-to-text format).

Following early pretraining, the field discovered what happens when neural language models are scaled aggressively. Five historical milestones established the foundational lessons of scale:

Historical Milestone	Year	Scale	Key Lesson
GPT-2	2019	1.5B	Fluent text generation; first signs of zero-shot task transfer
Kaplan Scaling Laws	2020	—	Smooth power-law predictability of training loss as a function of compute, data, and parameters
GPT-3	2020	175B	In-context learning via prompting without gradient updates
PaLM	2022	540B	Massive parameter scale, but substantially undertrained for its compute budget
Chinchilla	2022	70B	Compute-optimal scaling: train smaller models on far more tokens ($D \approx 20N$)

Every model in the table above is a **historical baseline**, not a current architecture. We will unpack Chinchilla's $D \approx 20N$ rule and its modern inference-time revisions when we reach Section 5.

Studying how these ideas evolved into modern systems is only possible because of open research and published technical reports. Openness in language modeling has progressed through distinct historical waves into a clear late-2026 taxonomy:

- **Early open replications (historical):** EleutherAI's The Pile dataset and GPT-J, Meta's OPT (175B parameters, plagued by hardware failures during training), and BigScience's BLOOM (176B).
- **Earlier open-weight waves (2023–2025, historical):** Llama 1, Llama 2 (2023), and Llama 3 (July 2024, trained on 15T tokens), Mistral and Mixtral, DeepSeek-V3 (December 2024; 671B total / 37B active parameters, 256 experts with 8 active + 1 shared) and DeepSeek-R1 (January 2025), Qwen3 (May 2025, 36T tokens), and Kimi K1.5 (January 2025) and Kimi K2 (July 2025, 384 experts with 8 active).
- **Current open-weight frontier (late 2026):** Models releasing weights and detailed technical reports whose performance approaches closed frontier models (GPT, Claude, Gemini).
- **Fully open-source models:** Projects releasing weights, paper, code, *and* training data. Prominent examples are **AI2 Olmo 2 / Olmo 3**, **NVIDIA Nemotron 3**, and **Marin** (which practices open development).

DEFINITION

Open-weight vs. fully open-source model: An **open-weight** model publicly releases trained parameter weights (typically alongside a technical report) but withholds training data or full training code, whereas a **fully open-source** model releases weights, paper, training code, and training data together.

COMMON MISCONCEPTION

You might think that any model with downloadable weights and an arXiv technical report is "fully open-source." That breaks whenever the training corpus is withheld: for example, NVIDIA **Nemotron 3.5** (October 2026) releases weights and a technical report but not its training data, making it an open-weight model rather than a fully open-source model like Nemotron 3 or Olmo 2/3.

Inspecting the **current late-2026 open-weight frontier** reveals a striking architectural convergence across independent labs:

Model	Released	Key Architectural & Training Highlights
DeepSeek-V4.1-Flash	Sept 2026	552B backbone parameters (16B active during decode / 8B active during prefill); 384 routed experts (6 active) + 1 shared expert; 1M context; 1 KV head shared by 64 query heads (`head_dim` 512); CSA2 sparse attention + cross-layer KV reuse + FP4 KV cache (890 bytes/token global KV); 3 multi-token prediction (MTP) layers; pretrained on 45T tokens
Kimi K3 / K3.1	July 2026	2.8T total / 104B active parameters; 896 routed experts (16 active) + 2 shared experts in a Stable LatentMoE layer (MXFP4 quantization-aware training); 1M context; 3 Kimi Delta Attention layers + 1 gated Multi-Head Latent Attention (MLA) layer per block; Attention Residuals; Per-Head Muon optimizer
GLM-5.3	2026	78 layers, hidden size 6144, 64 heads; MLA (`kv_lora_rank` 512, RoPE dim 64) + DSA sparse attention (top-k 2048); 256 experts (8 active + 1 shared); 1 MTP layer
Qwen3.8-Flash-Next	Aug 2026	125B total / 6B active parameters; 3:1 ratio of Gated DeltaNet to full attention (replaced by QSA sparse attention during continued pretraining); 512 experts (10 active + 1 shared); 51B-parameter n-gram embedding table kept in host RAM
MiniMax-M3	June 2026	Blockwise top-k MiniMax Sparse Attention built on Grouped-Query Attention (GQA); matches full GQA quality on a 109B model while cutting per-token attention compute by $28.4\times$ at 1M context
Xiaomi MiMo-V2.6	Oct 2026	Late-2026 open-weight frontier release (alongside DeepSeek-V4,

Model	Released	Key Architectural & Training Highlights
		Nemotron 3.5 Super/Lightning, and Ling/Ring 2.6)

Look closely at the shared 2026 architectural pattern across these systems: **fine-grained sparse Mixture of Experts (MoE) + hybrid linear/sparse attention + 1M-token context lengths + multi-token prediction (MTP)**, trained on tens of trillions of tokens (such as DeepSeek-V4.1-Flash's 45T tokens).

WORKED EXAMPLE

Why do late-2026 frontier models invest so heavily in sparse routing, compressed KV caches, and sparse attention? Plug in the exact late-2026 specifications to see the scale of the resource savings.

1. **Active vs. total parameters (DeepSeek-V4.1-Flash):** Out of 552B backbone parameters, each generated token activates only 16B parameters during decoding and 8B during prefill:

$$\frac{16\text{B}}{552\text{B}} \approx 2.9\% \text{ (decode)}, \qquad \frac{8\text{B}}{552\text{B}} \approx 1.45\% \text{ (prefill)}$$

The model stores half a trillion parameters of knowledge while paying the per-token arithmetic cost of an 8B–16B model.

2. **Global KV cache at 1M context (DeepSeek-V4.1-Flash):** Combining CSA2 sparse attention, cross-layer KV reuse, and FP4 KV quantization shrinks the global KV footprint to 890 bytes/token. Across a full 1M-token (10^6-token) context window, the global KV cache for an entire sequence totals:

$$890 \text{ bytes/token} \times 10^6 \text{ tokens} = 890 \times 10^6 \text{ bytes} = 890 \text{ MB}$$

Keeping a million-token context under one gigabyte of global KV memory makes long-context serving practical.

3. **Sparse attention speedup at 1M context (MiniMax-M3):** At 1M context on a 109B model, blockwise top-k MiniMax Sparse Attention cuts per-token attention compute by $28.4\times$ relative to dense GQA—reducing 28.4 units of attention compute to 1.0 unit while matching full GQA quality.

Why did the entire frontier converge on 1M-token contexts and aggressive inference compression? The answer lies in how the operational specification of a language model has evolved. Picture a four-stage left-to-right arrow diagram tracking what a language model *is* to its user:

- **2018 (BERT):** Something you **fine-tune** on a downstream task.
- **2020 (GPT-3):** Something you **prompt** to elicit in-context behavior.
- **2022 (ChatGPT):** Something you **talk to** in interactive dialogue.
- **2026 (Agents):** Something that **acts autonomously** over extended environments.

While the underlying technical fundamentals remain the same—attention, recurrence, kernels, distributed training, and optimization—the system specifications have shifted sharply. Autonomous agents operating across entire codebases and multi-turn tool trajectories require much longer contexts (up to 1M tokens). Moreover, because agents generate long reasoning chains and execute many tool calls —and because modern post-training relies heavily on generating reinforcement learning rollouts— **inference efficiency now dominates both deployment cost and RL training cost**.

This raises an immediate architectural question: what exact components inside a Transformer have changed between 2017 and 2026 to meet these specifications, and how do those components interact during training?

KEY IDEA

Late-2026 open-weight frontier models (such as DeepSeek-V4.1-Flash, Kimi K3, GLM-5.3, Qwen3.8-Flash-Next, and MiniMax-M3) have converged on fine-grained sparse MoE, hybrid linear and sparse attention, 1M-token contexts, and multi-token prediction—driven by the shift from static fine-tuning to autonomous agents where long contexts and inference efficiency dominate.

✎ Check Your Understanding

1. **Fill in the blank:** A model that releases its weights and technical report, but withholds its training data (such as Nemotron 3.5), is classified as an _____ model rather than a fully open-source model.
   Answer: open-weight. Fully open-source models (such as Olmo 2/3, Nemotron 3, and Marin) release weights, paper, code, and training data.
2. **Quick computation:** Kimi K3 has 2.8T $(2{,}800\text{B})$ total parameters and 104B active parameters per token. Approximately what percentage of its total parameters are active for any single token?
   Answer: $104/2{,}800 \approx 3.7\%$. Sparse MoE routing allows Kimi K3 to maintain 2.8T parameters of total capacity while activating fewer than 4% per token.

Section 3: Blueprint of a Language Model — Architecture and Training Dynamics

Constructing a language model from scratch spans **five technical pillars**, arranged in a five-box sequential roadmap from left to right:

1. **Basics:** Tokenization, Architecture, and the Training loop (the first unit we will examine in full depth).
2. **Systems:** Kernels, Parallelism, and Inference.
3. **Scaling:** Scaling recipes, IsoFLOP laws, and μP hyperparameter transfer.
4. **Data:** Evaluation, Filtering & deduplication, and Mixing & synthetic data.
5. **Alignment:** Weak supervision, PPO / DPO, and GRPO & RL.

We first take a structured tour through all five pillars so that every engineering detail fits into the complete system, beginning with Pillar 1's architecture and training dynamics.

The 2017 Transformer remains the conceptual backbone of modern language models, yet nearly every sub-component has been refined to improve stability, long-context memory footprint, and compute efficiency by late 2026:

Component	Original Transformer (2017)	Current Refinements (Late 2026)
Activation	ReLU	SwiGLU
Position	Sinusoidal	RoPE (Rotary Position Embeddings)
Normalization	Post-norm LayerNorm	Pre-norm RMSNorm, QK-Norm
Attention	Full multi-head attention	GQA, MLA, sparse / local attention (CSA2, DSA, QSA, MiniMax)
Hybrid sequence layers	None	Mamba-2, Gated DeltaNet, Kimi Delta Attention interleaved with attention
MLP	Dense FFN	Fine-grained sparse Mixture of Experts (MoE) + shared experts

Every row of this table addresses a concrete failure mode of the 2017 design:

- **Activation and position:** SwiGLU empirically outperforms ReLU for a given parameter budget, while RoPE encodes relative token positions cleanly and extends better to long contexts than fixed sinusoidal embeddings.
- **Normalization:** Placing normalization *after* the residual addition (post-norm LayerNorm) makes deep Transformer training fragile to gradient explosion or vanishing. Moving normalization *before* each sub-layer (pre-norm RMSNorm) simplifies computation and stabilizes gradient flow, while **QK-Norm** normalizes queries and keys before their dot product to prevent attention logits from blowing up.
- **Attention and hybrid sequence layers:** Full multi-head attention stores separate keys and values for every head and attends to every prior token, causing KV-cache memory and $O(L^2)$ attention compute to explode at 1M tokens. **Grouped-Query Attention (GQA)** shares KV heads across multiple query heads; **Multi-Head Latent Attention (MLA)** compresses keys and values into a low-rank latent vector; **sparse attention** (CSA2, DSA, QSA, MiniMax) restricts each token to attending over a selected subset of past positions; and **hybrid architectures** interleave recurrent/linear-time sequence layers (such as Mamba-2, Gated DeltaNet, or Kimi Delta Attention) with attention layers.
- **MLP:** Replacing a single dense feed-forward network with a **fine-grained sparse Mixture of Experts (MoE)** plus always-on **shared experts** decouples total model capacity from per-token compute.

DEFINITION

Fine-grained sparse Mixture of Experts (MoE): A feed-forward layer partitioned into many small specialized sub-networks (experts) where a learned router activates only a tiny subset of routed experts per token—often alongside one or more always-active shared experts—so total parameter count grows without proportionally increasing per-token FLOPs.

Crucially, architecture shape—hidden dimension, depth, head count, and expert count—is no longer chosen in isolation; it is **co-designed with hardware memory and KV-cache budgets**.

Turning an architecture into a trained model requires five tightly coupled training ingredients:

- **Loss function:** Standard next-token cross-entropy loss (recall that cross-entropy measures $-\log p(x_{t+1} \mid x_{\leq t})$, penalizing low predicted probability on the true next token), augmented in late-2026 models with **multi-token prediction (MTP)** heads that predict multiple future tokens from the same hidden representation (for instance, DeepSeek-V4.1-Flash uses 3 MTP layers and GLM-5.3 uses 1 MTP layer).
- **Optimizers: AdamW** remains the standard baseline, joined in late-2026 frontier runs by second-order and matrix-structured optimizers such as **SOAP** and **Muon** (used in DeepSeek-V4, and Per-Head Muon in Kimi K3).
- **Initialization and learning-rate schedules:** Careful initialization (Xavier or μ**P**, which we will revisit under Scaling Laws) paired with **cosine** or **WSD (warmup-stable-decay)** learning-rate schedules.
- **Regularization and batching:** Weight decay, dropout, and choosing batch sizes relative to the **critical batch size** (beyond which larger batches yield diminishing returns in optimization progress per token).
- **MoE stability: Auxiliary-loss-free load balancing**, which keeps tokens distributed evenly across experts without adding a competing auxiliary penalty term that degrades the language modeling loss.

All architectural and training choices sit inside a **core design tension**, visualized as an equilateral triangle with one property at each vertex and the statement **"Every choice balances all three"** at its center:

1. **Expressivity** (top vertex): the representational capacity of the model to capture complex patterns.
2. **Stability** (bottom-left vertex): keeping activation and gradient norms inside the "Goldilocks zone"—neither exploding to NaN nor collapsing to zero.
3. **Efficiency** (bottom-right vertex): training speed and inference speed per unit of hardware.

COMMON MISCONCEPTION

You might think that Expressivity, Stability, and Efficiency can be optimized independently—first maximizing expressivity, then tuning efficiency afterward. That breaks because nearly every architectural change moves all three vertices simultaneously: adding sparse MoE experts boosts expressivity and preserves per-token FLOP efficiency, yet immediately introduces a routing stability problem if tokens collapse onto a handful of experts.

WORKED EXAMPLE

Trace this three-way tension concretely using GLM-5.3's MoE configuration: 256 routed experts (with 8 active per token) plus 1 always-active shared expert, for 257 total experts per layer.

- **Per-token active fraction:** Each token computes through $8 + 1 = 9$ experts, which is:

$$\frac{9}{257} \approx 3.5\%$$

of the layer's expert parameters. Compared to a dense MLP holding the same total parameter count, per-token MLP FLOPs drop by roughly $257/9 \approx 28.6\times$ (a massive **efficiency** win while retaining 257-expert **expressivity**).

- **Failure without load balancing:** Suppose routing becomes unstable and collapses so that 100% of tokens select the exact same 8 routed experts. Every token still computes 9 experts, so per-token FLOPs look unchanged, yet 248 of the 256 routed experts receive zero tokens and zero gradient updates. The effective parameter capacity collapses from 257 experts down to 9, wiping out the expressivity gain.
- **Resolution:** Auxiliary-loss-free load balancing dynamically adjusts per-expert routing biases to steer load evenly across all 256 experts, maintaining **stability** without corrupting the primary cross-entropy gradient.

Designing an expressive, stable, and mathematically FLOP-efficient architecture is still only half the battle. A model with fewer theoretical FLOPs can still run slowly in practice if its operations stall waiting on memory transfers or inter-GPU communication. That brings us directly to Pillar 2: Systems.

KEY IDEA

Modern 2026 architectures replace almost every 2017 Transformer component—moving to SwiGLU, RoPE, pre-norm RMSNorm with QK-Norm, compressed/sparse/hybrid attention, and fine-grained sparse MoE—because every architectural and training decision must jointly balance expressivity, stability (keeping norms in the Goldilocks zone), and training/inference efficiency.

✎ Check Your Understanding

1. **What would happen if** an MoE router collapses and sends every token to the same small subset of experts instead of distributing tokens evenly?
   Answer: Although per-token FLOPs remain unchanged, the unused experts receive no tokens and no gradient updates, collapsing the model's effective expressivity to only the active subset.
2. **Yes or No:** Does replacing post-norm LayerNorm with pre-norm RMSNorm and QK-Norm address training stability? Briefly explain why.
   Answer: Yes. Pre-norm RMSNorm stabilizes gradient flow through deep residual paths, and QK-Norm constrains query and key magnitudes so attention logits stay inside a stable numerical range.

Section 4: Systems Engineering — Compute, Memory Movement, Parallelism, and Inference

An architecture's theoretical operation count tells you how much arithmetic is required, not how fast that arithmetic executes on real hardware. To understand real-world speed and cost, we start with **resource accounting**. For a standard dense Transformer, the total floating-point operations required for training are approximated by:

$$C = 6ND$$

Examine each symbol in this equation:

- C is the total training compute measured in floating-point operations (FLOPs).
- N is the number of active model parameters.
- D is the total number of training tokens processed.
- The factor of 6 accounts for both the forward pass and the backward pass per token per parameter: each parameter participates in roughly 2 FLOPs (a multiply and an add) during the forward pass and 4 FLOPs during the backward pass (computing gradients with respect to both activations and weights), summing to 6 FLOPs per parameter per token.

Taken as a single conceptual unit, $C = 6ND$ translates model size and dataset size directly into a total compute bill.

However, FLOPs alone do not determine runtime. Visualize a single GPU as a two-block vertical hardware diagram:

- **Top block — Compute (SMs):** Streaming Multiprocessors equipped with fast on-chip SRAM and tensor cores. For an NVIDIA B200, peak BF16 throughput is $2.25\ \text{PFLOP/s}$ ($2.25 \times 10^{15}\ \text{FLOP/s}$).
- **Bottom block — GPU Memory (HBM):** High-Bandwidth Memory storing the model weights, activations, KV cache, and optimizer states.
- **Bidirectional arrows between them:** Data must be loaded from HBM up to the SMs before arithmetic happens and written back down to HBM afterward, across a memory bus running at $8\ \text{TB/s}$ ($8 \times 10^{12}\ \text{bytes/s}$).

This separation between fast compute and comparatively slower HBM creates two distinct performance regimes, analyzed using **roofline analysis**.

> **DEFINITION**
> **Roofline analysis:** A performance model that compares an operation's arithmetic intensity (FLOPs performed per byte transferred from HBM) against the hardware's peak compute-to-bandwidth ratio to determine whether the operation is **compute-bound** (limited by arithmetic throughput) or **memory-bound** (limited by HBM data movement).

Benchmarking and profiling tools such as **Nsight** verify which bottleneck actually limits a workload in practice.

WORKED EXAMPLE

Apply both $C = 6ND$ and roofline analysis directly to the numbers above.

1. **Training compute bill:** Suppose you train a $N = 70\text{B}$ (7×10^{10}) parameter model on $D = 1\text{T}$ (10^{12}) tokens:

$$C = 6 \times (70 \times 10^9) \times 10^{12} = 420 \times 10^{21} = 4.2 \times 10^{23} \text{ FLOPs}$$

2. **Hardware compute-to-bandwidth balance point (NVIDIA B200):** Divide peak BF16 compute throughput by HBM bandwidth:

$$\frac{2.25 \times 10^{15} \text{ FLOP/s}}{8 \times 10^{12} \text{ bytes/s}} = 281.25 \text{ FLOP/byte}$$

This threshold of 281.25 FLOP/byte is the crossover between regimes. If a kernel loads a tensor from HBM and performs only 2 FLOP/byte (such as a simple elementwise activation), it uses less than 1% of the tensor cores' arithmetic capacity—it is severely **memory-bound**, spending almost all its time waiting on HBM. Only operations that reuse loaded data heavily enough to exceed 281.25 FLOP/byte (such as large matrix multiplications) become **compute-bound**.

Because moving data between HBM and on-chip SRAM is so expensive, the central principle of GPU **kernel engineering** is to **minimize data movement**. Compare two horizontal pipelines illustrating execution of two sequential operations, **Compute A** followed by **Compute B**:

- **Naive (2 kernels, 4 HBM trips):** Each primitive PyTorch operation launches a separate GPU kernel. Data flows `HBM -> Compute A -> HBM -> Compute B -> HBM`. Even though Compute B immediately needs the output of Compute A, the intermediate tensor is written out to HBM and read right back in, incurring four HBM transfers.
- **Fused (1 kernel, 2 HBM trips):** Data flows `HBM -> Compute A + B (on-chip) -> HBM`. The intermediate result stays inside fast on-chip SRAM while Compute B runs, cutting HBM traffic in half.

Two core kernel strategies exploit on-chip SRAM in this way:

DEFINITION

Operator fusion: Combining multiple sequential operations (such as a matrix multiplication followed by an activation function) into a single GPU kernel so intermediate tensors stay on-chip instead of making redundant round-trips to HBM.

DEFINITION

Tiling: Partitioning large matrices or attention score blocks into small tiles that fit inside fast on-chip SRAM so all intermediate reductions occur on-chip before writing final outputs to HBM—the core mechanism behind FlashAttention.

To implement fused and tiled operations beyond standard framework primitives, engineers write custom kernels using **CUDA**, **Triton**, **CUTLASS**, and **CuTe DSL**.

WORKED EXAMPLE

Quantify the wall-time speedup of operator fusion on a memory-bound pair of elementwise operations A and B acting on a $16\ \text{MB}$ tensor on an $8\ \text{TB/s}$ ($8 \times 10^6\ \text{MB/s}$) HBM bus:

- **Naive (2 kernels, 4 HBM trips):** Kernel A reads $16\ \text{MB}$ and writes $16\ \text{MB}$; Kernel B reads $16\ \text{MB}$ and writes $16\ \text{MB}$. Total HBM traffic is $4 \times 16\ \text{MB} = 64\ \text{MB}$, taking:

$$\frac{64\ \text{MB}}{8 \times 10^6\ \text{MB/s}} = 8\ \mu\text{s}$$

- **Fused (1 kernel, 2 HBM trips):** One read of $16\ \text{MB}$ and one write of $16\ \text{MB}$ total $32\ \text{MB}$, taking:

$$\frac{32\ \text{MB}}{8 \times 10^6\ \text{MB/s}} = 4\ \mu\text{s}$$

Without changing a single floating-point operation, operator fusion achieves a $2\times$ speedup purely by eliminating two unnecessary HBM trips.

When training scales beyond a single device to **1,024+ GPUs**, the exact same governing principle reappears at a steeper cliff: inter-GPU links are slower than on-chip HBM, so distributed training must **minimize inter-GPU communication**. Distributed systems shard memory (parameters, activations, gradients, and optimizer states) and coordinate across devices using **collective communication primitives**—specifically `gather`, `reduce`, and `all-reduce`—along **five axes of parallelism**:

Parallelism Dimension	What Is Split	Primary Goal
Data	Batch examples (and sharded optimizer states / weights)	Scale throughput and cut per-GPU memory state
Tensor	Matrix rows and columns within a single layer	Fit wide layers across fast intra-node links
Pipeline	Groups of layers across sequential stages	Scale model depth with point-to-point activation sends
Sequence	Token positions along the context length	Support long contexts (e.g., 1M tokens)
Expert	MoE experts across separate devices	Scale sparse parameter count cleanly

Finally, once a model is trained, **inference** powers deployment, evaluation, test-time compute, and reinforcement learning rollouts. Inference divides into two structurally opposite phases, illustrated by a two-box diagram connected by an arrow:

- **Phase 1: Prefill (Compute-Bound):** All prompt tokens $t_1, t_2, t_3, \ldots, t_L$ are already known and processed **in parallel**. Because all L tokens share the same loaded model weights in a single batched matrix multiplication, arithmetic intensity is high and the hardware operates in the compute-bound regime.
- **Phase 2: Decode (Memory-Bound):** Output tokens $y_1 \rightarrow y_2 \rightarrow y_3$ must be generated **one token at a time sequentially**, because y_2 depends on y_1 and y_3 depends on y_2. Every generated token requires streaming the active model weights and the accumulated KV cache out of HBM just to process a single vector, starving the SMs and locking decode into the memory-bound regime.

> **COMMON MISCONCEPTION**
> You might think that generating a 1,000-token response costs the same per token as reading a 1,000-token prompt because both run through the same Transformer layers. That breaks because prefill amortizes loading the model weights from HBM across all $1{,}000$ prompt tokens simultaneously (compute-bound), whereas decode reloads the weights from HBM for every single generated token sequentially (memory-bound).

Three families of engineering techniques accelerate memory-bound decoding:

1. **Cheaper models: Pruning** (removing parameters), **quantization** (representing weights or KV cache in lower precision), and **distillation** (training a smaller model to match a larger teacher).
2. **Speculative decoding:** A small, cheap draft model proposes several candidate tokens sequentially, and the full target model verifies all proposed tokens in a single parallel forward pass—turning memory-bound sequential steps into compute-bound verification while preserving the **exact** output probability distribution of the target model.
3. **Serving systems:** Fused kernels and **continuous batching** (dynamically grouping tokens from multiple concurrent requests so weight loads from HBM are amortized across many users at once).

DEFINITION

Speculative decoding: An inference acceleration method in which a lightweight draft model proposes multiple candidate future tokens that the full target model verifies in parallel in one forward pass, speeding up memory-bound decoding while mathematically preserving the target model's exact output distribution.

Knowing how to extract maximum speed from hardware still leaves a major strategic question unanswered: before launching a massive training run, how do you choose the model size, token count, and hyperparameters so you do not waste a 10^{25}-FLOP budget on a suboptimal configuration? That brings us to Pillar 3: Scaling Laws, followed by Data and Alignment.

KEY IDEA

Training compute scales as $C = 6ND$, yet real hardware speed is governed by the boundary between compute throughput and data movement—between HBM and SMs on a single GPU (tackled via operator fusion and tiling), across GPUs (tackled via five-axis parallelism), and between compute-bound prefill and memory-bound decode during inference (tackled via compression, continuous batching, and exact speculative decoding).

✎ Check Your Understanding

1. **Quick computation:** Using $C = 6ND$, how many FLOPs are required to train a 10B-parameter dense model on 200B (2×10^{11}) tokens?
   Answer: $C = 6 \times (10^{10}) \times (2 \times 10^{11}) = 1.2 \times 10^{22}$ FLOPs.
2. **Yes or No:** Does speculative decoding trade away accuracy by approximating the target model's output distribution? Explain briefly.
   Answer: No. Because the full target model verifies the draft model's proposed tokens in parallel and accepts or rejects them according to the target probabilities, speculative decoding preserves the exact output distribution of the target model.

Section 5: Scaling Laws, Data Pipelines, and Alignment

Suppose you are handed a target budget of 10^{25} FLOPs and asked to produce the strongest possible language model. Tuning model width, depth, learning rate, and token count by trial and error at 10^{25} FLOPs is financially and practically impossible—you get essentially one shot.

Pillar 3 (Scaling Laws) solves this through a conceptual shift: instead of tuning a single run at target scale, you design a **scaling recipe**—a deterministic mapping from compute budget to hyperparameters ($\text{FLOPs} \rightarrow \text{hyperparameters}$). You execute smaller experiments across lower compute scales (for

example, up to 10^{24} FLOPs), fit a scaling law to the resulting losses, optimize the recipe, and predict the target loss *before* launching the final run. At frontier scale, **predictability** is at least as important as theoretical optimality.

Scaling laws do not hold automatically under arbitrary neural network setups. If optimal learning rates drift unpredictably as model width increases, small-scale experiments cannot predict large-scale behavior. Reliable extrapolation requires parameterizations that guarantee **hyperparameter transfer**, such as **μP (Maximal Update Parameterization)**.

> **DEFINITION**
> **Hyperparameter transfer (μP):** A model initialization and learning-rate scaling parameterization designed so that hyperparameters (such as learning rate) tuned on a small model remain optimal when the model is scaled up to much larger widths.

The most famous scaling trade-off asks how to split a fixed compute budget $C = 6ND$ between a bigger model (N) and more training tokens (D). To find the optimal allocation, researchers construct **IsoFLOP curves** (introduced in Kaplan 2020 and refined in Chinchilla 2022). Visualize a plot with **Model parameters N (log scale)** on the horizontal axis and **Training loss** on the vertical axis:

- For each fixed compute budget—C_1 (smallest, leftmost), C_2 (middle), and C_3 (largest, rightmost)—we train several models of different sizes N, setting $D = C/(6N)$ so every model on a given curve spends the exact same FLOPs.
- Each budget traces a downward-opening **U-shaped parabola**: if N is too small (left arm), the model lacks capacity and loss stays high despite seeing many tokens; if N is too large (right arm), the model is undertrained on too few tokens and loss again rises.
- Each parabola reaches a unique minimum at an optimal parameter count $N^*(C)$. Connecting the minima across C_1, C_2, C_3 traces a smooth dashed downward-sloping line, the $N^*(C)$ **fit**, which extrapolates straight to the target compute budget.

> **DEFINITION**
> **IsoFLOP curve:** A plot of training loss versus model parameter count N at a fixed total compute budget $C = 6ND$, whose U-shaped minimum identifies the compute-optimal model size $N^*(C)$ for that budget.

Chinchilla's empirical fit yielded a clean rule of thumb for compute-optimal training:

$$D \approx 20N$$

Read this relation directly: to minimize training loss for a fixed training FLOP budget, you should train on roughly 20 tokens per parameter.

COMMON MISCONCEPTION

You might think that every frontier lab should strictly obey $D \approx 20N$ because it minimizes training loss for a fixed training compute budget. That breaks once you account for **inference serving cost**: Chinchilla optimizes *training* FLOPs alone and ignores the cost of serving millions of users afterward. Modern models deliberately **over-train** smaller active parameter counts far past $20N$ (for example, DeepSeek-V4.1-Flash trains on 45T tokens with 16B decode-active parameters) so the deployed model is much cheaper to serve at inference time.

WORKED EXAMPLE

Solve the compute-optimal allocation explicitly and compare it against an inference-optimized over-trained model at the exact same training budget $C = 5.88 \times 10^{23}$ FLOPs.

1. **Chinchilla compute-optimal allocation ($D = 20N$):** Substitute $D = 20N$ into $C = 6ND$:

$$C = 6N(20N) = 120N^2 \implies N^* = \sqrt{\frac{5.88 \times 10^{23}}{120}} = \sqrt{4.9 \times 10^{21}} = 7 \times 10^{10} = 70\text{B}$$

The corresponding token budget is:

$$D^* = 20N^* = 20 \times (7 \times 10^{10}) = 1.4 \times 10^{12} = 1.4\text{T tokens}$$

2. **Over-trained for cheaper inference ($D = 200N$, $10\times$ more tokens per parameter):** Under the identical training compute budget $C = 5.88 \times 10^{23}$ FLOPs:

$$C = 6N(200N) = 1{,}200N^2 \implies N = \sqrt{\frac{5.88 \times 10^{23}}{1{,}200}} = \sqrt{4.9 \times 10^{20}} \approx 22.1\text{B parame}$$

$$D = 200N \approx 4.43\text{T tokens}$$

By shifting away from the Chinchilla training-loss minimum, you obtain a model over $3\times$ smaller at serving time for the same training FLOP budget—dramatically cutting HBM memory and decode latency across its deployment lifetime.

A scaling law is only as meaningful as the metric on its vertical axis and the tokens fed into the training loop. This brings us to **Pillar 4: Evaluation and Data**.

Evaluation serves two distinct engineering purposes that place different requirements on metrics:

- **Internal evaluation** guides model iteration during research and requires **smoothness across scales** and reliable relative ordering between candidate runs.
- **External evaluation** measures real-world utility and requires **ecological validity** and absolute quality.

Two core evaluation families anchor these goals: **perplexity** measured on private, non-web documents (preventing benchmark contamination from web crawls), and **frontier capability benchmarks** such as **GPQA**, **HLE**, **SWE-Bench**, and **Terminal-Bench**. Because language models are general-purpose systems—

expected to handle multilingual text, conversation, and agentic coding—no single metric suffices; evaluation demands a diverse suite, and what we want the model to do dictates both evaluation design and data collection.

Raw data scraped from web crawls, books, arXiv, and GitHub (which also raises copyright fair-use questions and prompts licensing deals such as with Reddit) is nowhere near ready for training. It passes through a **six-stage data processing pipeline**, moving left to right:

1. **Raw Sources** (HTML, PDF, code) →
2. **Transform** (extract clean text from markup and documents) →
3. **Filter** (remove low-quality and unsafe content) →
4. **Deduplicate** (using algorithms such as **Bloom filters** and **MinHash** to save compute and reduce verbatim memorization) →
5. **Mix** (optimizing domain proportions with methods such as **RegMix** and **OlMix**) →
6. **Rewrite** (generating **synthetic** or rewritten text, such as **WRAP**, to align raw documents with downstream tasks).

Training then organizes this curated data into **three curriculum stages**: **pretraining** (large and diverse), **mid-training** (high-quality and long-context data), and **post-training** (supervised fine-tuning conversations and agentic tool-calling traces).

Finally, **Pillar 5 (Alignment and Post-Training)** bridges the gap between a base model trained to predict the next token and an assistant or autonomous agent that follows instructions and solves complex tasks reliably. Pretraining relies on **full supervision** (every token has an exact target label). Alignment transitions to **weak supervision** because for hard reasoning, coding, or safety tasks, **it is often far easier to critique or verify an output than to generate a perfect demonstration from scratch**.

> **DEFINITION**
> **Weak supervision (in post-training):** Training a model using scalar feedback, pairwise preferences, or automated verification scores on complete model responses rather than token-by-token ground-truth demonstrations.

Alignment operates as a three-step closed loop moving left to right:

1. **Generate Responses:** Sample candidate rollouts from the current model.
2. **Score Responses:** Evaluate each rollout using a **human** rater, an automated **verifier** (such as unit tests or math checkers), or an **LM judge**.
3. **Update Policy:** Adjust the model parameters to increase the probability of higher-scoring outputs and decrease the probability of lower-scoring ones.

Three core optimization algorithms drive step 3: **PPO** (classic online reinforcement learning with a learned value function), **DPO** (offline optimization directly from preference pairs), and **GRPO** (online RL that removes the separate value function by scoring groups of rollouts relative to one another). Late-2026 frontier pipelines chain these stages sequentially: **SFT → asynchronous RL / RLVR (RL with Verifiable Rewards) → on-policy distillation**. For instance, DeepSeek-V4.1-Flash runs SFT, then asynchronous RL,

then on-policy distillation from more than 40 teachers; Nemotron 3.5 runs SFT, then RLVR (using router replay, an FP32 LM head, and a global batch of 8,192 with 16 rollouts per prompt), then expert teachers and multi-teacher on-policy distillation (MOPD). The central engineering challenges in post-training are **RL instability** and the **systems trade-off between asynchronous rollout throughput and on-policyness** (generating rollouts asynchronously keeps GPUs busy, but updates calculated from stale policies can destabilize training).

Our tour across all five pillars is complete. Now we return to the very first boundary of Pillar 1—where raw human text enters the neural network: **Tokenization**.

KEY IDEA

Scaling laws turn frontier training into a predictable engineering discipline through scaling recipes, μP hyperparameter transfer, and IsoFLOP curves ($D \approx 20N$, modified in practice by over-training for cheaper inference). Curated multi-stage data pipelines and weak-supervision post-training loops (SFT $\rightarrow$ async RL/RLVR $\rightarrow$ on-policy distillation) then convert raw compute into reliable, aligned capabilities.

✎ Check Your Understanding

1. **Quick computation:** According to the Chinchilla compute-optimal rule ($D \approx 20N$), approximately how many training tokens should be used to train a 5B-parameter dense model?
   Answer: $D \approx 20 \times 5\text{B} = 100\text{B}$ tokens.
2. **Fill in the blank:** In Pillar 4, measuring perplexity on _____ documents prevents benchmark contamination from web-crawled training data.
   Answer: private, non-web. Evaluating on documents excluded from public web crawls ensures the score reflects generalization rather than memorization.

Section 6: Tokenization Fundamentals and the Failure of Naive Schemes

A neural language model cannot operate directly on raw Unicode strings—its embedding table and matrix multiplications require discrete integers. **Tokenization** is the interface that converts arbitrary raw text into the discrete atoms a language model consumes and converts generated integers back into text.

DEFINITION

Tokenization: The deterministic translation between raw input text (Unicode strings or byte streams) and sequences of discrete integer token IDs, defined by paired `encode()` and `decode()` operations.

Visualize this interface as a three-box horizontal pipeline:

- On the left sits a **Unicode String** (for example, `"Hello, 🌍! 你好!"`).
- A forward arrow labeled **`encode()`** maps the string to the middle box, a **Token Sequence (ints)** (such as `[13225, 11, 130321, ...]`), while a reverse arrow labeled **`decode()`** maps the integer sequence back to the original Unicode string.
- From the token sequence, a forward arrow feeds into the right box, the **Language Model**, which defines a probability distribution $p(x_1, \ldots, x_L)$ over integer sequences of length L.

In software, any tokenizer implements this minimal two-method contract:

```
class Tokenizer(ABC):
    def encode(self, string: str) -> list[int]: ...
    def decode(self, indices: list[int]) -> str: ...
```

A strict correctness requirement governs this interface: **`encode()` and `decode()` must round-trip losslessly for arbitrary Unicode strings**—that is, `decode(encode(string)) == string`.

Beyond merely converting text into numbers, tokenization delivers two critical efficiency wins that tie directly back to Section 1's efficiency lens: it **shrinks sequence length** (for example, turning $1{,}000$ raw bytes into roughly 250 tokens), and it **allocates more model capacity per byte to complex or rare chunks** while collapsing predictable strings into single tokens.

To see a modern production tokenizer in action, inspect OpenAI's `o200k_base` tokenizer (via `tiktoken`) on the multilingual string `"Hello, 🌍! 你好!"`, which occupies **20 UTF-8 bytes**:

```
tokenizer = tiktoken.get_encoding("o200k_base")
string = "Hello, 🌍! 你好!"                    # 20 UTF-8 bytes
indices = tokenizer.encode(string)
# [13225, 11, 130321, 235, 0, 220, 177519, 0]    # 8 tokens
assert tokenizer.decode(indices) == string
```

Laying out those **8 token IDs** side by side as labeled boxes reveals surprising internal structure:

`[13225]` `[11]` `[130321]` `[235]` `[0]` `[220]` `[177519]` `[0]`

Notice two striking features highlighted in that breakdown:

- A single 4-byte emoji (🌍) does *not* become a single token; it splits across **two byte-level tokens** (`130321` and `235`).
- Conversely, the two-character Chinese greeting `"你好"` (6 UTF-8 bytes) merges completely into **one single token** (`177519`).

These behaviors expose four **practical quirks of subword tokenizers** that directly explain many puzzling behaviors of language models—including spelling errors, digit arithmetic mistakes, and sensitivity to trailing spaces:

1. **Leading whitespace attaches to the word:** `" world"` is typically encoded as one single token rather than `" "` followed by `"world"`.

2. **Position changes representation:** In `"hello hello"`, the first `"hello"` (no leading space) and the second `" hello"` (with a leading space) map to completely different integer token IDs.
3. **Numbers split into arbitrary digit chunks:** Multi-digit numbers are grouped every few digits based on corpus frequency rather than aligned to decimal place value (units, tens, hundreds).
4. **Multilingual and rare-symbol asymmetry:** Frequent scripts pack many bytes into each token, whereas rare scripts and symbols fall back to multi-token byte fragments, paying a higher token tax per unit of meaning.

Why do modern tokenizers behave this way instead of simply splitting on characters, bytes, or words? Answering that requires formalizing the **core trade-off of tokenization**: **compression ratio versus vocabulary size** $|V|$.

DEFINITION

Compression ratio: The average number of raw UTF-8 bytes represented by each token produced by a tokenizer, measuring how aggressively the tokenizer shortens input sequences.

$$\text{compression ratio} = \frac{\text{UTF-8 bytes}}{\text{number of tokens}}$$

Break this ratio apart term by term:

- The numerator, UTF-8 bytes, is the byte length of the raw text when encoded in UTF-8 (a fixed property of the input string, independent of the tokenizer).
- The denominator, number of tokens, is the length L of the integer list returned by `encode()`.

As a single conceptual unit, the compression ratio tells you the factor by which tokenization shrinks sequence length L relative to raw bytes. Higher compression is critical because standard Transformer self-attention scales as $O(L^2)$: shrinking L yields a quadratic reduction in pairwise attention interactions.

However, raising compression by making the vocabulary size $|V|$ larger triggers two severe penalties:

1. **Data sparsity:** Rare tokens appear too infrequently during training to receive enough gradient updates to learn strong representations.
2. **Parameter overhead:** Both the input embedding matrix and the final output-projection (LM head) matrix scale **linearly with** $|V|$ (each has dimension $|V| \times d$).

WORKED EXAMPLE

Compute both the compression ratio and the attention savings on two concrete inputs.

1. **Multilingual string under `o200k_base`:** `"Hello, 🌍! 你好!"` contains 20 UTF-8 bytes and encodes into 8 tokens:

$$\text{compression ratio} = \frac{20 \text{ bytes}}{8 \text{ tokens}} = 2.5 \text{ bytes/token}$$

2. **Attention savings on standard English text:** Suppose a tokenizer compresses a $1{,}000$-byte document into 250 tokens:

$$\text{compression ratio} = \frac{1{,}000}{250} = 4.0 \text{ bytes/token}$$

Because sequence length L drops by a factor of 4, the L^2 term in standard attention drops from $1{,}000^2 = 1{,}000{,}000$ pairwise token interactions down to $250^2 = 62{,}500$—a $16\times$ **reduction** in pairwise attention work.

We can plot every tokenization strategy on a two-dimensional scatter chart with **Vocabulary size $|V|$ (log scale)** on the horizontal axis and **Compression (bytes / token)** on the vertical axis. Three naive paradigms land in undesirable corners of this plane, exposing exactly why each fails.

Approach 1: Character-Based Tokenization

The simplest idea is mapping each Unicode character directly to its integer code point via Python's `ord()` and reversing with `chr()`:

```python
class CharacterTokenizer(Tokenizer):
    def encode(self, string: str) -> list[int]:
        return list(map(ord, string))
    def decode(self, indices: list[int]) -> str:
        return "".join(map(chr, indices))
```

For example, `ord("a") == 97` and `ord("🌍") == 127757`. Character tokenization round-trips cleanly, yet it suffers from **two fatal flaws**:

1. **Huge vocabulary:** Unicode defines roughly 150K **characters**, bloating the embedding and output matrices.
2. **Severe sparsity and low compression:** Rare code points like 🌍 rarely appear in training, and compression remains poor—on `"Hello, 🌍! 你好!"` (20 UTF-8 bytes, 13 characters), compression is only $20/13 \approx 1.54$ bytes/token (and for pure ASCII English, it is exactly 1.0 byte/token).

On the trade-off plot, Character sits at the bottom-right—large $|V|$ (~150K) and low compression—making it the **worst of both worlds**.

Approach 2: Byte-Based Tokenization

Instead of Unicode characters, encode any Unicode string into its UTF-8 byte stream (values in $0 \ldots 255$) and use raw byte values as token IDs:

```python
class ByteTokenizer(Tokenizer):
    def encode(self, string: str) -> list[int]:
        return list(map(int, string.encode("utf-8")))
    def decode(self, indices: list[int]) -> str:
        return bytes(indices).decode("utf-8")
```

In UTF-8, `"a"` encodes to 1 byte (`b"a"`), while `"🌍"` encodes to 4 bytes (`b"\xf0\x9f\x8c\x8d"`). Byte tokenization gains a huge advantage: a tiny, fixed vocabulary of $|V| = 256$ with **zero out-of-vocabulary tokens**—any Unicode string in any language is representable.

Its fatal flaw sits directly on the vertical axis of the trade-off plot (bottom-left at $|V| = 256$): its **compression ratio is always exactly** 1.0 byte/token. Sequences become far too long for $O(L^2)$ attention layers. Still, byte-level encoding gives us a crucial gift we will build on in the next section: a **universal base alphabet**.

Approach 3: Word-Based Tokenization

To achieve strong compression, split text into words and punctuation using a regular expression:

```python
string = "I'll say supercalifragilisticexpialidocious!"
chunks = regex.findall(r"\w+|.", string)
# ['I', "'", 'll', ' ', 'say', ' ',
#  'supercalifragilisticexpialidocious', '!']  # 8 tokens, 44 bytes
```

Word tokenization achieves semantically meaningful units and high compression ($44/8 = 5.5$ $\text{bytes}/\text{token}$ above), landing at the upper-right of the trade-off plot. Yet it breaks on **three fatal flaws**:

1. **Unbounded vocabulary:** Every distinct word, typo, identifier, or number in a corpus demands its own token ID.
2. **Rare-word sparsity:** Long-tail words appear too rarely to learn useful embeddings.
3. **`<UNK>` tokens:** Any word unseen during vocabulary construction collapses into a catch-all unknown token (`<UNK>`), which destroys lossless round-tripping and invalidates perplexity comparisons across tokenizers.

> **DEFINITION**
> **Out-of-vocabulary (`<UNK>`) token:** A special placeholder ID assigned to any input string absent from a fixed vocabulary, collapsing distinct unseen words into one symbol and preventing lossless reconstruction.

COMMON MISCONCEPTION

You might think that replacing rare or unseen words with `<UNK>` is merely a minor cosmetic nuisance during decoding. That breaks evaluation fundamentally: a tokenizer that maps many hard, rare words to `<UNK>` makes the language model's prediction task artificially easier—it only has to predict the single symbol `<UNK>` rather than the exact word—producing misleadingly low perplexity scores that cannot be compared fairly across models.

WORKED EXAMPLE

Compare all three naive approaches head-to-head on `"Hello, 🌍! 你好!"` (20 UTF-8 bytes: 7 bytes for `"Hello, "`, 4 bytes for `"🌍"`, 2 bytes for `"! "`, 3 bytes each for `"你"` and `"好"`, and 1 byte for `"!"`; 13 characters total):

- **Character:** Produces 13 tokens. Compression $= 20/13 \approx 1.54$ bytes/token, $|V| \approx 150\text{K}$. Fails on both vocabulary bloat and sparsity.
- **Byte:** Produces 20 tokens. Compression $= 20/20 = 1.00$ bytes/token, $|V| = 256$. Guaranteed lossless, but $2.5\times$ longer than `o200k_base` (8 tokens), inflating $O(L^2)$ attention work by $2.5^2 = 6.25\times$.
- **Word:** Requires every distinct token (including `"Hello"`, `"🌍"`, `"你好"`) to exist in a pre-built word table. If `"你好"` or `"🌍"` was absent from the training vocabulary, it collapses to `<UNK>`, making `decode(encode(string))` fail to recover the original string.

Summarizing the four paradigms reveals the exact gap waiting to be filled:

Method	Vocab Size $\lvert V\rvert$	Compression	Unseen Text?	Main Failure Mode
Character	~150K	Low (~1–1.5)	Yes (Unicode)	Huge sparse vocab + short chunks
Byte (UTF-8)	256	1.0	Always	Sequences too long for $O(L^2)$ attention
Word	Unbounded	High	Needs `<UNK>`	Huge vocab, rare words, broken perplexity
Byte-level BPE	**Chosen ($\lvert V\rvert$)**	**High (~4)**	**Always**	**Separate heuristic step (non-learned)**

On the trade-off scatter plot, there is an open **sweet spot** in the upper-middle region: moderate, user-chosen vocabulary size $|V|$ combined with high compression (around 4 bytes/token) and guaranteed coverage of any unseen text. Reaching that sweet spot requires **training** a subword vocabulary directly from data—which brings us to **Byte-Pair Encoding (BPE)**.

KEY IDEA

Tokenization balances compression ratio ($\text{UTF-8 bytes/tokens}$, critical for $O(L^2)$ attention) against vocabulary size $|V|$ (which drives sparsity and embedding matrix size). Character, byte, and word schemes each fail on at least one of sequence length, vocabulary bloat, or `<UNK>` lossiness, motivating a learned byte-level subword vocabulary.

✎ Check Your Understanding

1. **Quick computation:** A tokenizer encodes a 500-byte UTF-8 document into 125 tokens. What is its compression ratio, and by what factor does this reduce L^2 pairwise attention interactions compared to raw byte tokenization?
   Answer: The compression ratio is $500/125 = 4.0\ \text{bytes/token}$. Because sequence length L shrinks by $4\times$, L^2 pairwise attention interactions drop by $4^2 = 16\times$.
2. **Yes or No:** Under a modern subword tokenizer like `o200k_base`, do the two occurrences of the letters `"hello"` in `"hello hello"` always map to the same token ID? Explain why.
   Answer: No. Because leading whitespace attaches to words in subword tokenizers, the first `"hello"` (no leading space) and the second `" hello"` (with a leading space) typically map to different token IDs.

Section 7: Byte-Pair Encoding (BPE) — Algorithm, Trace, and Encoding/Decoding

To combine the universal lossless coverage of UTF-8 bytes with the high compression of words—at a fixed, controllable vocabulary size $|V|$—we learn the vocabulary directly from corpus statistics using **Byte-Pair Encoding (BPE)**. Originally introduced by Gage (1994) for general data compression, BPE was adapted to neural machine translation by Sennrich et al. (2016) and made **byte-level** in GPT-2 (2019).

DEFINITION

Byte-Pair Encoding (BPE): A data-driven, greedy subword tokenization algorithm that starts from the 256 single-byte tokens and iteratively merges the most frequently occurring adjacent pair of tokens into a new token ID until reaching a desired vocabulary size.

Because BPE starts from raw UTF-8 bytes, frequent byte sequences automatically merge into common words and subwords, while rare or unseen strings stay decomposed into shorter subwords or raw bytes—**never** needing an `<UNK>` token.

The greedy BPE training procedure consists of three steps:

1. **Initialize** the vocabulary with all 256 single-byte tokens ($0 \ldots 255$).

2. **Count** every adjacent pair of tokens across the training corpus.
3. **Merge** the single most frequent adjacent pair into a new token ID ($256 + i$), replace all occurrences of that pair in the corpus with the new ID, and repeat for `num_merges` iterations.

After `num_merges` iterations, the final vocabulary size is exactly $|V| = 256 + \text{num_merges}$. Here is the complete training implementation:

```python
@dataclass(frozen=True)
class BPETokenizerParams:
    vocab: dict[int, bytes]            # index -> bytes
    merges: dict[tuple[int, int], int] # (id1, id2) -> new_id

def train_bpe(string: str, num_merges: int) -> BPETokenizerParams:
    indices = list(map(int, string.encode("utf-8")))
    merges: dict[tuple[int, int], int] = {}
    vocab: dict[int, bytes] = {x: bytes([x]) for x in range(256)}
    for i in range(num_merges):
        counts = count_adjacent_pairs(indices)
        pair = max(counts, key=counts.get)
        new_index = 256 + i
        merges[pair] = new_index
        vocab[new_index] = vocab[pair[0]] + vocab[pair[1]]
        indices = merge(indices, pair, new_index)
    return BPETokenizerParams(vocab=vocab, merges=merges)
```

Notice two subtle mechanics in this implementation:

- `vocab[new_index] = vocab[pair[0]] + vocab[pair[1]]` concatenates the byte strings of the two merged tokens. Every token ID—whether a single byte or a multi-merge word—maps directly to an exact `bytes` object.
- The helper `merge(indices, pair, new_index)` scans **left-to-right** and replaces every non-overlapping occurrence of `pair` with `new_index`. On a repeated sequence like `"aaa"` (`[97, 97, 97]`), merging `(97, 97) -> 256` left-to-right replaces the first two bytes and advances past both, yielding `[256, 97]`.

WORKED EXAMPLE

Trace `train_bpe("the cat in the hat", num_merges=3)` step by step, reconstructing all four rows of the worked trace. The input `"the cat in the hat"` has **18 UTF-8 bytes** (writing `␣` for the space byte `32`):

- **Row 0 — Init (18 tokens, compression ratio $18/18 = 1.00$):**

`[t] [h] [e] [␣] [c] [a] [t] [␣] [i] [n] [␣] [t] [h] [e] [␣] [h] [a] [t]`

Count adjacent pairs: `(t, h)` appears 2 times (positions 0 and 11), `(h, e)` appears 2 times, `(e, ␣)` appears 2 times, `(a, t)` appears 2 times, and all other pairs appear once. With four pairs tied at count 2, Python's `max(counts, key=counts.get)` selects the first encountered maximum: `(116, 104)` (`"t"`, `"h"`).

- **Row 1 — Merge 1: `(116, 104)` `"t"+"h"` → `256` (`"th"`), count $= 2 \implies 16$ tokens:**

`[th]` `[e]` `[␣]` `[c]` `[a]` `[t]` `[␣]` `[i]` `[n]` `[␣]` `[th]` `[e]` `[␣]` `[h]` `[a]` `[t]`

Recount adjacent pairs: `(256, 101)` (`"th"`, `"e"`), `(101, 32)` (`"e"`, `"␣"`), and `(97, 116)` (`"a"`, `"t"`) each appear 2 times. The first encountered maximum is `(256, 101)`.

- **Row 2 — Merge 2: `(256, 101)` `"th"+"e"` → `257` (`"the"`), count $= 2 \implies 14$ tokens:**

`[the]` `[␣]` `[c]` `[a]` `[t]` `[␣]` `[i]` `[n]` `[␣]` `[the]` `[␣]` `[h]` `[a]` `[t]`

Recount adjacent pairs: `(257, 32)` (`"the"`, `"␣"`) and `(97, 116)` (`"a"`, `"t"`) each appear 2 times. The first encountered maximum is `(257, 32)`.

- **Row 3 — Merge 3: `(257, 32)` `"the"+"␣"` → `258` (`"the␣"`), count $= 2 \implies 12$ tokens (compression ratio $18/12 = 1.50$):**

`[the␣]` `[c]` `[a]` `[t]` `[␣]` `[i]` `[n]` `[␣]` `[the␣]` `[h]` `[a]` `[t]`

Observe the hierarchical structure: Merge 2 consumes token `256` created by Merge 1, and Merge 3 consumes token `257` created by Merge 2. Each merge builds higher-level tokens out of previously merged tokens, growing bottom-up from raw bytes.

Once `BPETokenizerParams` (`vocab` and `merges`) have been learned, encoding new text and decoding integer sequences back into strings require only a few lines:

```python
class BPETokenizer(Tokenizer):
    def __init__(self, params: BPETokenizerParams):
        self.params = params

    def encode(self, string: str) -> list[int]:
        indices = list(map(int, string.encode("utf-8")))
        for pair, new_index in self.params.merges.items():
            indices = merge(indices, pair, new_index)
        return indices

    def decode(self, indices: list[int]) -> str:
        bytes_list = list(map(self.params.vocab.get, indices))
        return b"".join(bytes_list).decode("utf-8")
```

Notice how `decode()` works: it looks up the byte string for each token ID in `vocab`, concatenates all byte chunks into one contiguous byte stream with `b"".join(bytes_list)`, and calls `.decode("utf-8")` once at the end. Concatenating before UTF-8 decoding is essential—remember from Section 6 that a single 4-byte emoji like 🌍 can be split across two separate tokens (`130321` and `235`), neither of which is valid UTF-8 on its own, yet their concatenated bytes decode cleanly back to `"🌍"`.

COMMON MISCONCEPTION

You might think that during `encode()`, the order in which you apply the rules in `self.params.merges` does not matter—or that you should simply greedily match the longest token in `vocab` first. Both ideas break BPE's hierarchical merge tree: a later merge rule such as `(256, 101) -> 257` can only match *after* the earlier rule `(116, 104) -> 256` has actually produced token `256`. Applying merges out of their learned priority order skips valid merges and produces a different, inconsistent tokenization.

WORKED EXAMPLE

Apply our 3 learned merges from `"the cat in the hat"` — `(116, 104) -> 256`, `(256, 101) -> 257`, and `(257, 32) -> 258` —to an unseen sentence `"the quick brown fox"` (**19 UTF-8 bytes**).

1. **Convert to raw UTF-8 byte IDs (19 tokens):**

`[116, 104, 101, 32, 113, 117, 105, 99, 107, 32, 98, 114, 111, 119, 110, 32, 102, 111, 120]`

2. **Apply Merge 1, `(116, 104) -> 256` (18 tokens):**

`[256, 101, 32, 113, 117, 105, 99, 107, 32, 98, 114, 111, 119, 110, 32, 102, 111, 120]`

3. **Apply Merge 2, `(256, 101) -> 257` (17 tokens):**

`[257, 32, 113, 117, 105, 99, 107, 32, 98, 114, 111, 119, 110, 32, 102, 111, 120]`

4. **Apply Merge 3, `(257, 32) -> 258` (16 tokens, compression ratio $19/16 = 1.1875$):**

`[258, 113, 117, 105, 99, 107, 32, 98, 114, 111, 119, 110, 32, 102, 111, 120]`

5. **Decode:** Look up each ID in `vocab` (`vocab[258] = b"the "`, `vocab[113] = b"q"`, `vocab[117] = b"u"`, …), concatenate into `b"the quick brown fox"`, and decode as UTF-8 to recover `"the quick brown fox"` losslessly. The unseen words `"quick"`, `"brown"`, and `"fox"` degrade gracefully into raw bytes with zero `<UNK>` tokens.

6. **Pitfall check (wrong merge order):** If you mistakenly ran Merge 2 `(256, 101) -> 257` and Merge 3 `(257, 32) -> 258` *before* Merge 1 `(116, 104) -> 256`, neither `(256, 101)` nor `(257, 32)` would exist in the raw byte stream. Only Merge 1 would fire at the end, leaving `[256, 101, 32, ...]` (18 tokens instead of 16).

Our toy `train_bpe` and `BPETokenizer` implement the exact mathematical core of BPE, yet running them verbatim on a real multi-terabyte corpus with a 200K-token vocabulary would be far too slow and would waste vocabulary slots on undesirable cross-boundary merges. Four concrete engineering upgrades bridge toy BPE to production tokenizers.

KEY IDEA

Byte-level BPE initializes with the 256 UTF-8 bytes and greedily merges the most frequent adjacent pair on each iteration, growing hierarchical subword tokens from the bottom up while guaranteeing lossless fallback to raw bytes on unseen text; during `encode()`, merges must be applied in their exact learned order.

✎ Check Your Understanding

1. **Quick trace:** Suppose a BPE tokenizer has learned only two merges: `(97, 98) -> 256` (`"a"+"b" -> "ab"`) and `(256, 99) -> 257` (`"ab"+"c" -> "abc"`). What token sequence does `encode("abcd")` produce (given ASCII `d = 100`)?
   Answer: Starting from `[97, 98, 99, 100]`, Merge 1 produces `[256, 99, 100]` and Merge 2 produces `[257, 100]`.
2. **What would happen if** `decode()` called `.decode("utf-8")` on each individual token's bytes before joining them, instead of concatenating all token bytes with `b"".join(...)` first?
   Answer: Decoding would crash with a UTF-8 error whenever a multi-byte character (such as a 4-byte emoji like 🌍) was split across two byte-level tokens, because individual byte fragments are not valid UTF-8 strings on their own.

Section 8: Production BPE Engineering and the Horizon Beyond Tokenization

Look closely at the naive `BPETokenizer` and `train_bpe` code from Section 7 and ask what breaks when scaling $|V|$ to 100K–200K tokens and corpora to trillions of bytes. Four specific bottlenecks and pathologies force production tokenizers to add four engineering upgrades:

1. **Avoid scanning all merges during `encode()`:** In the naive loop `for pair, new_index in self.params.merges.items()`, encoding a 10-character string iterates through every single merge rule in a 200K-entry table, even though almost none of those pairs appear in the string. Production encoders invert the loop: they examine only the adjacent pairs **actually present** in the current token sequence, pick the present pair with the **lowest merge rank** (the one learned earliest during training), merge it, and repeat until no adjacent pair in the sequence exists in `merges`.
2. **Pre-tokenization (e.g., GPT-2 regex):** Running BPE directly across raw unbroken text allows merges to cross word and punctuation boundaries. Common nouns followed by different punctuation marks get learned as redundant separate tokens—for example, `"dog."` and `"dog!"` consume separate vocabulary slots even though both represent the same word plus punctuation. Production tokenizers first apply a **pre-tokenization** step (such as the GPT-2 regular expression) that splits raw text into coarse chunks; BPE then counts pairs and executes merges strictly *within* each chunk, forbidding merges across chunk boundaries.

DEFINITION

Pre-tokenization: A coarse rule-based splitting step (typically a regular expression) applied before BPE training and encoding to partition raw text into chunks so that BPE merges cannot cross word, whitespace, or punctuation boundaries.

3. **Special tokens:** Language models rely on control delimiters—such as `<|endoftext|>`—to mark document boundaries, conversation turns, or tool calls. If `<|endoftext|>` passed through regex splitting and ordinary BPE, it would be fragmented into subwords (`<`, `|`, `end`, `of`, `text`, `|`, `>`) or merged with neighboring text. Production tokenizers detect and reserve **special tokens** intact *before* pre-tokenization or merging.

DEFINITION

Special token: A reserved control string (such as `<|endoftext|>`) assigned a fixed dedicated token ID and matched atomically prior to pre-tokenization and BPE merging so it is never split or merged with surrounding text.

4. **Fast training via incremental pair-count updates:** In the naive `train_bpe` loop, `count_adjacent_pairs(indices)` rescans the entire corpus from start to finish on every single merge iteration—an $O(\text{corpus size} \times \text{num_merges})$ disaster. Yet merging a pair `(a, b) -> ab` only alters adjacent pairs **immediately touching** the occurrences of `(a, b)`. Production trainers maintain pair-count indices incrementally around merged sites, updating only the affected local neighbors at each step.

COMMON MISCONCEPTION

You might think that pre-tokenization guarantees a tokenizer can never combine a space with a word. That breaks because GPT-2-style pre-tokenization regexes deliberately attach a single leading space to the following word chunk (`" dog"`), while splitting *punctuation* away into its own chunk (`"."`, `"!"`). That exact design decision is why `" world"` becomes one token (Section 6) while `"dog."` and `"dog!"` do not duplicate `"dog"`.

WORKED EXAMPLE

Trace two of these production upgrades concretely.

1. **Pre-tokenization on `"dog. dog! dog?"`:**
   - *Without pre-tokenization:* BPE scans across punctuation and observes repeated sequences `d-o-g-.`, `d-o-g-!`, and `d-o-g-?`, potentially minting distinct tokens `"dog."`, `"dog!"`, and `"dog?"` that fragment capacity across three separate embeddings.
   - *With pre-tokenization:* The regex first splits the string into coarse chunks `["dog", ".", " dog", "!", " dog", "?"]`. BPE runs inside each chunk independently, so `"dog"` (or `" dog"`) merges cleanly without ever fusing with `"."`, `"!"`, or `"?"`.
2. **Incremental count update during Merge 1 of `"the cat in the hat"`:**

- Before Merge 1, the two occurrences of `(t, h)` sit inside `t h e` (positions 0–2) and `␣ t h e` (positions 10–13).
- Replacing `(t, h) -> th` only touches the immediate boundary pairs at those two sites:
  - Decrement `(t, h)` by 2 (from $2 \to 0$).
  - Decrement `(h, e)` by 2 (from $2 \to 0$) and `(␣, t)` by 1 (from $1 \to 0$).
  - Increment `(th, e)` by 2 (from $0 \to 2$) and `(␣, th)` by 1 (from $0 \to 1$).
- Every pair elsewhere in `" cat in "` and `" hat"` — `(c, a)`, `(a, t)`, `(i, n)`, `(h, a)` —is completely untouched and never recounted. On a billion-token corpus, updating only local neighborhoods makes BPE training orders of magnitude faster.

Even with all four production upgrades in place, a deep conceptual tension remains: **BPE is an effective, data-driven heuristic, yet it is still a separate, non-learned preprocessing stage** frozen before neural network training even begins. All of the tokenization quirks from Section 6—leading-space sensitivity, arbitrary digit chunking, and script asymmetry—stem from this rigid boundary between tokenizer and neural model.

The long-term research dream is **end-to-end tokenizer-free models** operating directly on raw bytes. A growing research lineage—including **ByT5**, **MegaByte**, **BLT**, **T-FREE**, and **H-Net**—explores architectures that learn byte grouping internally. These ideas are promising, though they have not yet scaled to replace subword tokenizers at the frontier.

Whatever future learned mechanism eventually replaces BPE must satisfy **two core properties**:

1. **Higher-level chunks (abstractions):** The model must operate internally on multi-byte abstractions rather than burning uniform quadratic compute at the raw byte level—a principle that generalizes across modalities including text, video, and DNA.
2. **Variable-length chunks:** Chunk boundaries must adapt dynamically so the model allocates more compute to harder, information-dense regions and less compute to predictable boilerplate.

Viewed through this lens, **tokenization is really a static, rule-based form of adaptive computation**.

With the full five-pillar landscape mapped and tokenization built from first principles through production BPE, we have established *what* integer sequences enter the Transformer and *why* efficiency governs every design choice. Next, we turn to detailed **resource accounting**—quantifying FLOPs, memory footprints, and bandwidth across every matrix multiplication and activation tensor inside the model.

KEY IDEA

Production BPE scales toy BPE by encoding in merge-rank order over present pairs, pre-tokenizing to block cross-punctuation merges, protecting special tokens intact, and updating pair counts incrementally around merged sites. Looking ahead, tokenizer-free models (ByT5, MegaByte, BLT, T-FREE, H-Net) aim to replace this static heuristic with learned, variable-length, higher-level chunking —treating tokenization as a form of adaptive computation.

✎ Check Your Understanding

1. **Fill in the blank:** Applying a regex split before BPE training so merges cannot cross word and punctuation boundaries (preventing duplicate tokens like `"dog."` and `"dog!"`) is called _____.
   Answer: pre-tokenization. Pre-tokenization partitions raw text into coarse chunks and restricts BPE merges to occur strictly within each chunk.
2. **Yes or No:** During fast BPE training, after merging a chosen pair `(a, b)` into a new token ID, do we need to recount adjacent pairs across the entire corpus? Explain why.
   Answer: No. Merging `(a, b)` changes adjacent pairs only at positions immediately neighboring the merged occurrences, so pair counts can be updated incrementally at those local sites.