LLM Workbook: Architecture and Breakthroughs Behind Modern Generative AI

Dr. Yves J. Hilpisch[1]

December 18, 2025 (work in progress)

[1]Get in touch: `https://linktr.ee/dyjh`. Web page `https://hilpisch.com`. Research, structuring, drafting, and visualizations were assisted by GPT 5.1 as a co-writing tool under human direction.

Contents

1 How to Use This Workbook

This workbook is a self-contained, hands-on companion to *A Short History of Computer Science: From Commodore C64 to Cloud AI*. It zooms in on the major breakthroughs, components, and modern architecture elements that make the frontier large language models (LLMs) catalogued in the December 2025 snapshot so powerful in practice. The tone is intentionally conversational, but every story is grounded in equations, schematic diagrams, and runnable code.

The primary reader is a curious student or practitioner who has seen at least one deep-learning model before—for example, a simple feed-forward network or a small Transformer—and now wants a "big picture" map of how contemporary LLMs are actually built and deployed. Mentors, instructors, and technically minded decision-makers can also use the workbook as a structured lab to connect the historical narrative of the main book with the engineering realities of present-day generative AI.

Instead of treating each architecture trick or hardware advance in isolation, the workbook keeps returning to a single question: *How does this particular idea (from positional encodings to mixed-precision training) change the computation graph, the tensors that flow through it, and the capabilities we see at the surface?* You will move back and forth between concept maps, simplified model diagrams, and small PyTorch snippets that crystallise these ideas.

You will get the most value if you treat the workbook as a studio rather than a lecture. Read a short section, then sketch the associated schematic figure on paper, annotate it with concrete dimensions (such as context length or number of heads), and, where code is provided, reproduce and extend it in your own Python environment.

December 2025 Frontier Snapshot

Frontier model snapshot (late 2025):

- OpenAI: GPT-5.2 as the latest frontier flagship for reasoning and long-context work.
- Google: Gemini 3 Pro (plus Gemini 3 Deep Think) as the current frontier generation, replacing 2.5 Pro/Flash.
- Anthropic: Claude Opus 4.5 as the top Claude frontier model.
- xAI: Grok 4.1 as its newest frontier model.
- Meta: Llama 4 (including large mixture-of-experts (MoE) variants) as its open frontier family.
- Mistral: Mistral Large / Mistral 3-series frontier models.

Target Reader and Prerequisites

The workbook assumes that you:

- are comfortable with basic linear algebra (vectors, matrices, matrix–vector products),
- can read and write elementary Python (variables, functions, loops, list comprehensions),
- have at least a rough picture of what a neural network and a Transformer block are,
- and are curious about how data, models, optimisation, and hardware come together in real LLM systems.

You *do not* need prior experience training large models yourself; the goal is to demystify how production-scale systems relate to the smaller examples you can run on a laptop.

What You Will Build

By the end of this workbook you will have:

- a concept map that links data pipelines, tokenisation, model architecture, optimisation, and hardware for modern decoder-only LLMs,
- a small, from-scratch Transformer block in PyTorch that mirrors the core building block of contemporary decoder-only models,
- schematic TikZ diagrams of the end-to-end LLM pipeline, the internal structure of a Transformer block, and the attention patterns that drive in-context learning,
- and a practical vocabulary for discussing scaling laws, context windows, parallelism strategies, and deployment trade-offs.

Suggested Learning Paths

Different readers can take different routes through the material:

- If you mainly care about *using* LLMs safely and effectively, focus on the high-level concept maps and the deployment and safety sections, skimming detailed training diagrams on a first pass.
- If you want to understand how Transformers achieve their capabilities, prioritise the sections on tokenisation, embeddings, attention, and the internal block diagrams.
- If you are a mentor or instructor, treat each schematic figure as the spine of a whiteboard session or live-coding demo, letting students fill in missing labels, shapes, and code fragments.

Whichever path you choose, keep a notebook—digital or paper—where you redraw the key TikZ schematics in your own hand. This practice turns otherwise abstract architecture labels into a concrete mental model you can reuse when new model families appear.

The rest of the workbook follows a simple rhythm: each major breakthrough or architecture component is introduced with a short narrative, visualised via a schematic diagram, and then reinforced with a small code experiment wherever feasible. Sections are written to stand on

their own, but the full story emerges when you see how the diagrams interlock.

2 Concept Map and Learning Goals

This section lays out the conceptual map of the workbook so you can always see how data, models, hardware, and deployment fit together. At the centre sits a generic "modern LLM" node; around it we place five perspectives: data and objectives, architecture and training, inference and tools, hardware and parallelism, and evaluation and safety. The concept map in Figure 1 is not meant to be memorised; it is a visual reminder of the conversations that will keep happening between these perspectives.

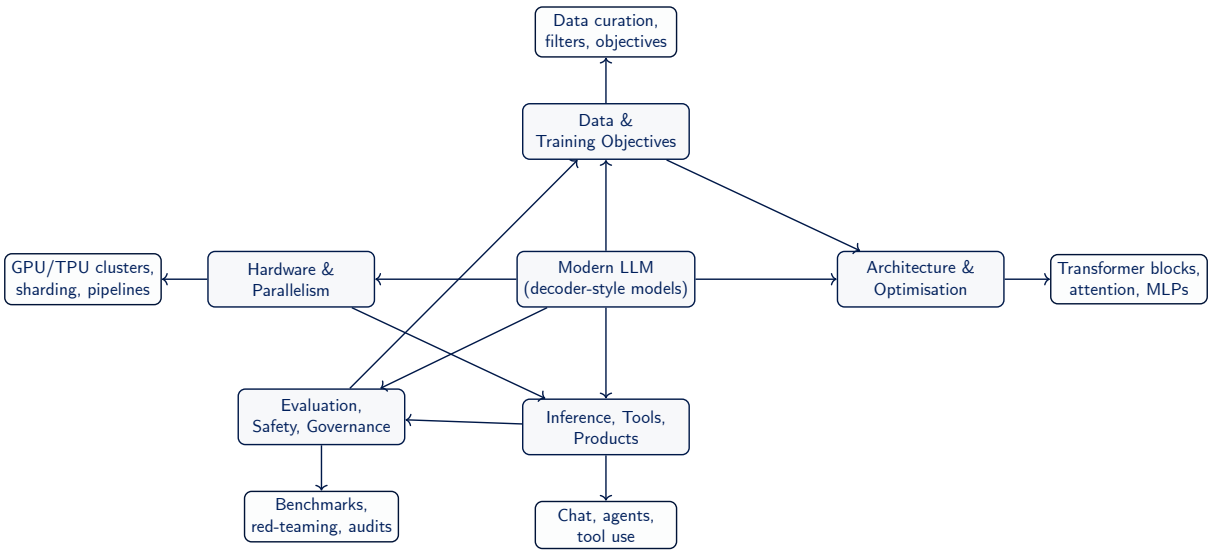

Figure 1: Concept map for the LLM workbook: a central "modern LLM" node connects data and training objectives, model architecture and optimisation, inference and tools, hardware and parallelism, and evaluation and safety. Each surrounding box corresponds to a major section of the workbook.

Starting at the top, the *Data and Training Objectives* node captures everything that happens before the first gradient step: corpus construction, filtering, deduplication, and the choice of pre-training and fine-tuning objectives. Moving clockwise, the *Architecture and Optimisation* node covers tokenisation, embeddings, positional encodings, stacked Transformer blocks, and the optimisation procedures that make them train in practice.

The *Hardware and Parallelism* node reminds us that the frontier LLMs listed in the December 2025 snapshot exist only because we can spread their parameters and activations across many graphics processing units (GPUs) and tensor processing units (TPUs). Below that, the *Inference, Tools, Products* node focuses on how raw model capabilities turn into chat interfaces, agents, and application programming interfaces (APIs). Finally, the *Evaluation, Safety, Governance* node anchors the discussion in real-world impact: how we test models, probe their limits, and design safeguards.

Learning Goals at a Glance

Across the whole workbook you will:

- build a layered mental model of modern LLMs that connects data, architecture, optimisation, hardware, and deployment,
- learn to read and draw schematic diagrams of Transformer blocks and end-to-end LLM pipelines,
- relate high-level marketing claims ("multimodal reasoning", "tool use", "agentic workflows") to specific components in these diagrams,
- and understand how incremental breakthroughs—from better optimisers to longer context windows—add up to the models you use every day.

Keep Figure 1 nearby as a map: each new section will light up one or more of its boxes and arrows.

3 Breakthroughs Behind Modern LLMs

Before diving into the pipeline details, it helps to zoom out and see which concrete breakthroughs made the frontier-class systems listed above possible. At a very high level, these models combine five ingredients: the Transformer architecture [1], massive datasets, scaling-aware training procedures [3, 4], alignment techniques such as reinforcement learning from human feedback (RLHF) [5], and clever inference-time engineering.

The first major shift was architectural: the move from recurrent networks to Transformers with self-attention. Recurrent networks process tokens one step at a time and struggle to remember long-range structure. Self-attention, by contrast, lets each position in the sequence look at every other position in parallel. This both improves gradient flow during training and maps cleanly to modern accelerators, where matrix multiplications are cheap.

The second breakthrough was empirical: scaling laws. When you increase parameter count, dataset size, and compute budget together, language models follow surprisingly smooth curves in terms of loss and downstream capability. This observation turned model design into an engineering discipline: rather than guessing a good size, you can target a particular scale based on available compute and data and expect reliable returns.

The third breakthrough concerns alignment and instruction following. Raw language models trained only to predict the next token can be powerful but also unruly. Modern systems add a second training phase in which models are fine-tuned on curated instruction-style data and then further nudged with RLHF or related techniques. Practically, this means the model learns not just to continue text, but to respond helpfully, follow constraints, and avoid some classes of harmful outputs.

The remaining breakthroughs live closer to the hardware: mixed-precision training, activation checkpointing, tensor and pipeline parallelism, and optimised attention kernels. At inference time, key-value (KV) caching, speculative decoding, and load-balanced serving reduce latency and cost, making real-time chat interfaces and APIs feasible.

Quick Timeline of Key Ideas

Roughly speaking, frontier LLMs build on:

- *Transformers and self-attention* as the core architecture for long-range dependency modelling,
- *Compute- and data-aware scaling* that treats model size, dataset size, and training budget as design variables,
- *Instruction tuning and RLHF* that turn raw generative models into helpful assistants,
- *Multimodality and tool use* that allow models to work with images, code, and external tools and APIs,
- and *Hardware-aware training and serving* that combine model parallelism, efficient kernels, and caching at scale.

The rest of the workbook shows where each of these ideas lives in the diagrams and code.

4 From Tokens to Representations

Large language models never see raw text directly. Instead, they see sequences of integers produced by a tokeniser and then mapped into dense vectors by an embedding layer. Figure 2 summarises this path from human-readable prompts to the tensors that enter the Transformer stack. The goal in this section is to make each intermediate object—from byte-pair encoding (BPE) [6] and SentencePiece tokenisation [7] to positional encodings—concrete.

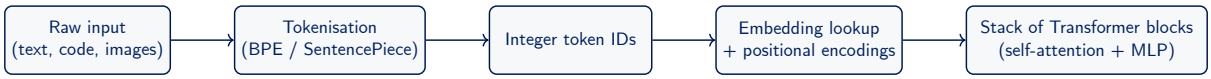

Figure 2: High-level view from user input to the internal Transformer stack. Each box hides concrete arrays: tokenisers turn strings into integer sequences; embedding layers map integers to vectors; positional encodings inject order information; and the resulting tensors feed a stack of Transformer blocks.

For a typical decoder-style model, a single user prompt might produce a tensor X of shape (B, L), where B is the batch size (often 1 during interactive use) and L is the number of tokens after tokenisation. After embedding lookup and positional encodings we obtain a tensor H_0 of shape (B, L, d_{model}) that flows into the first Transformer block. All later layers keep this shape but change the values.

Shapes from Prompt to Embeddings

To keep the pipeline concrete, imagine:

- a batch size $B = 2$ (two prompts processed together),
- a tokenised length $L = 128$ after padding or truncation,
- an embedding dimension $d_{\text{model}} = 4096$ typical of large LLMs.

The tensors then look like this:

- token IDs: $X \in \{0, \ldots, V-1\}^{B \times L}$, where V is the vocabulary size,
- embeddings: $E \in \mathbb{R}^{V \times d_{\text{model}}}$,
- embedded sequence: $H_0 = \text{Embed}(X) + P \in \mathbb{R}^{B \times L \times d_{\text{model}}}$, where P holds positional encodings broadcast along the batch dimension.

Once you can picture these three tensors, later diagrams for attention and transformer blocks become much easier to parse.

Minimal PyTorch Sketch

Here is a tiny PyTorch example that mirrors Figure 2:

Token IDs to embeddings

```
In [1]: import torch

In [2]: B, L, V, d_model = 2, 8, 50_000, 1024

In [3]: token_ids = torch.randint(0, V, (B, L))

In [4]: embed = torch.nn.Embedding(num_embeddings=V, embedding_dim=d_model)

In [5]: H0 = embed(token_ids)  # shape: (B, L, d_model)
In [6]: H0.shape
Out[6]: torch.Size([2, 8, 1024])
```

On real systems the same pattern holds, just with larger L and d_{model}, additional modalities, and more careful handling of padding and masking.

5 Inside a Transformer Block

Modern LLMs are built by stacking many copies of a relatively small computation pattern: the Transformer block. Figure 3 shows a "pre-norm" variant, where each sub-layer (attention or MLP) is preceded by layer normalisation and wrapped in a residual connection. The important idea is that all blocks share the same input and output shape; internally they modify the representation while preserving its dimensions.

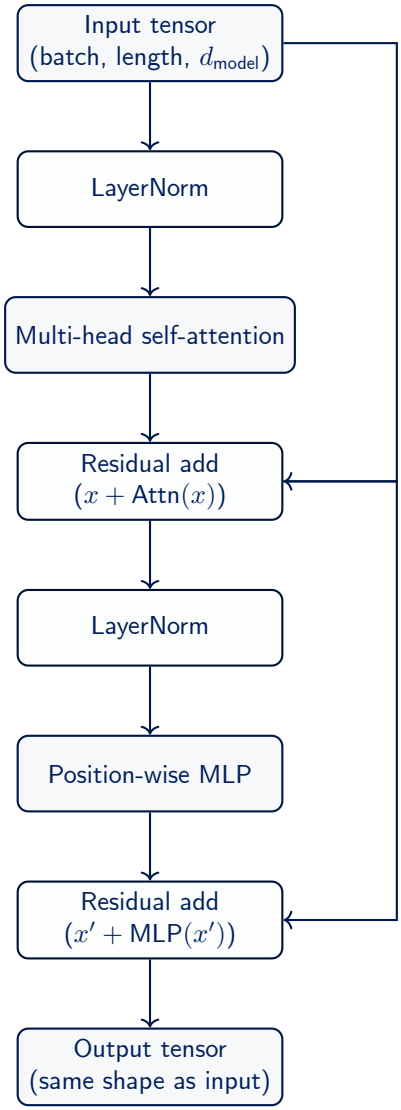

Figure 3: Schematic pre-norm Transformer block: LayerNorm, multi-head self-attention, and a position-wise multi-layer perceptron (MLP), each wrapped in a residual connection.

If we zoom in mathematically, each block implements a function

$$H_{\ell+1} = H_\ell + \mathrm{MLP}\big(\mathrm{LN}_2(H_\ell + \mathrm{Attn}(\mathrm{LN}_1(H_\ell)))\big),$$

where H_ℓ and $H_{\ell+1}$ have shape (B, L, d_{model}), Attn is multi-head self-attention, MLP is a two- or three-layer feed-forward network applied at each position, and $\mathrm{LN}_1, \mathrm{LN}_2$ are layer normalisations. Residual connections allow gradients to flow directly from later layers back to earlier ones, making very deep stacks trainable.

A Minimal PyTorch Block

One way to connect Figure 3 to code is to sketch a small Transformer block:

Toy Transformer block in PyTorch

```python
import torch
import torch.nn as nn

class TinyBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_mlp):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.attn = nn.MultiheadAttention(d_model, n_heads, batch_first=True)
        self.ln2 = nn.LayerNorm(d_model)
        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_mlp),
            nn.GELU(),
            nn.Linear(d_mlp, d_model),
        )

    def forward(self, x):  # x: (B, L, d_model)
        h = self.ln1(x)
        attn_out, _ = self.attn(h, h, h)  # self-attention
        x = x + attn_out
        h = self.ln2(x)
        x = x + self.mlp(h)
        return x
```

Even though frontier-class models contain many such blocks and additional features, the basic skeleton remains very close to this example.

6 Attention as a Grid of Interactions

Self-attention is the workhorse that lets LLMs connect information across long contexts. To reason about its behaviour it helps to look at attention as a matrix of interactions between positions in the input sequence. Each row of this matrix tells you which earlier tokens a given position is currently "looking at" when updating its representation. Figure 4 summarises the query, key, value, and attention-weight matrices that make up this grid.

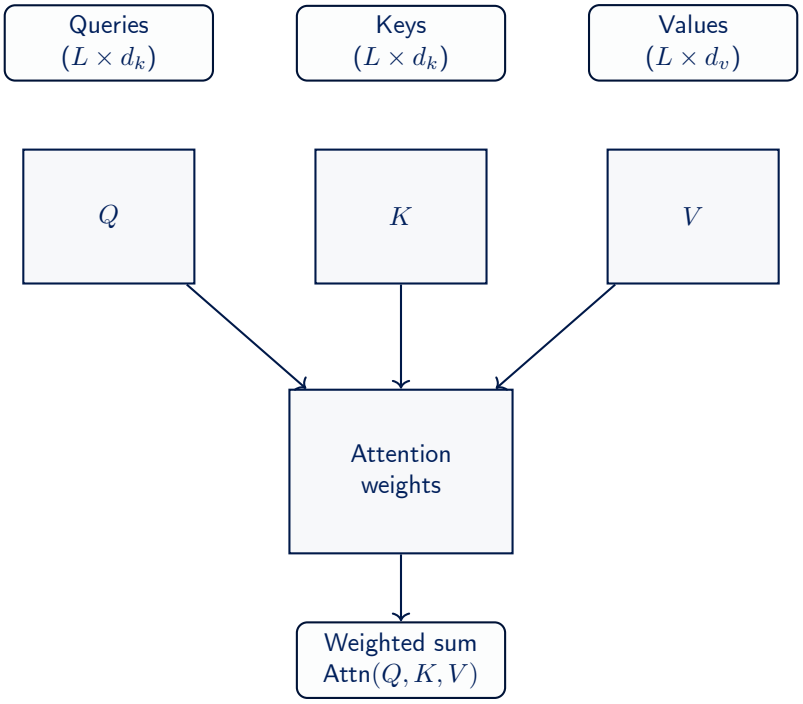

Figure 4: Schematic view of self-attention as a grid of interactions between positions in a sequence, highlighting the role of queries (Q), keys (K), values (V), and attention weights.

In a typical single-head, self-attention layer we start from $H \in \mathbb{R}^{B\times L\times d_{\text{model}}}$ and learn three projection matrices $W_Q, W_K, W_V \in \mathbb{R}^{d_{\text{model}}\times d_k}$. We form

$$Q = HW_Q, \qquad K = HW_K, \qquad V = HW_V,$$

so $Q, K \in \mathbb{R}^{B\times L\times d_k}$ and $V \in \mathbb{R}^{B\times L\times d_v}$. The attention scores are then

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

where the matrix product is taken over the feature dimension, M encodes causal or padding masks, and the softmax acts row-wise to turn each row of scores into a probability distribution over keys. Finally, the output is $O = AV$, which has the same sequence length L but mixes value vectors according to the learned attention patterns.

For a concrete feel, imagine a short prompt such as "The cat naps" so $L = 3$ and each token contributes a single-dimensional query/key score. One possible scaled dot-product before masking is

$$\frac{QK^\top}{\sqrt{d_k}} \approx \begin{bmatrix} 3.0 & 1.0 & 0.2 \\ 1.1 & 2.5 & 0.5 \\ 0.0 & 0.3 & 1.8 \end{bmatrix}.$$

Applying the row-wise softmax gives

$$A \approx \begin{bmatrix} 0.83 & 0.13 & 0.04 \\ 0.18 & 0.66 & 0.16 \\ 0.03 & 0.08 & 0.89 \end{bmatrix},$$

so the first row mostly copies the first token, the second row blends the first two tokens while still glancing at the third token, and the third row stays close to the most recent token (each row sums to 1 because of the normalised softmax).

When we enforce autoregressive causality, we add a mask matrix M that sends any future key score to $-\infty$; for three tokens the mask looks like

$$M_{\text{causal}} = \begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{bmatrix}.$$

After adding M_{causal} to the raw dot-products, softmax can only place mass on the current or earlier keys. Row one becomes locked to the first column and row two cannot use the third column at all, giving a triangular pattern in the attention grid. Figure 5 visualises the resulting non-causal and causal attention matrices for this toy example.

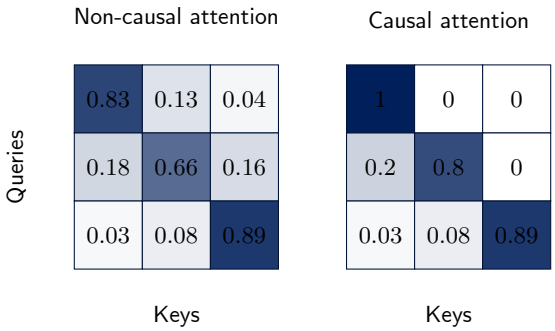

Figure 5: Toy attention heatmaps for a 3-token prompt: without a mask the grid can be dense, but an autoregressive mask zeros out future entries.

The heatmaps in Figure 5 make this contrast explicit: the non-causal grid retains dense weights while the masked grid zeros out future positions.

Technical Details: Scaled Dot-Product Attention

When you see "multi-head self-attention" in a model description, you can read it as:

- *Per head:* compute Q, K, V by linear projection, form the scaled dot products $QK^\top/\sqrt{d_k}$, apply masking and softmax, and then mix the values V.
- *Across heads:* repeat this process h times with different learned projections, concatenate the resulting $O^{(1)}, \ldots, O^{(h)}$ along the feature axis, and apply one final linear projection back to d_{model}.
- *Shapes:* if H has shape (B, L, d_{model}) and there are h heads, a common design is $d_k = d_v = d_{\text{model}}/h$, giving attention weight matrices of shape (B, h, L, L) and outputs of shape (B, L, d_{model}).

Thinking of attention as a learnable, data-dependent averaging over positions helps explain why LLMs are so good at tracking long-range structure in code, text, and other modalities.

7 Scaling, Parallelism, and Hardware

Training and serving frontier LLMs is as much a hardware engineering challenge as it is a modelling problem. Models with hundreds of billions of parameters cannot fit onto a single GPU or TPU; even storing the optimiser states and intermediate activations requires careful sharding across many devices. Figure 6 sketches the layers of this hardware stack.

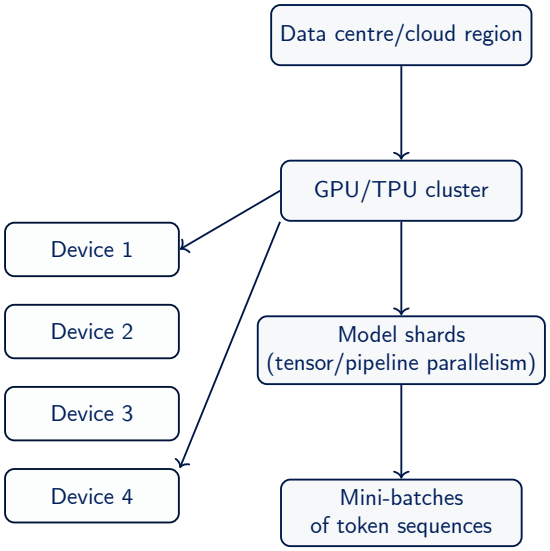

Figure 6: Schematic hardware stack for training and serving large language models, from data centre region down to individual accelerator devices.

At a coarse level, three forms of parallelism are common:

- *Data parallelism:* each device (or device group) holds a full copy of the model but processes different mini-batches of data; gradients are averaged across replicas.
- *Tensor model parallelism:* individual weight matrices are split across devices along their row or column dimensions so that a single layer can be computed in parallel.
- *Pipeline parallelism:* the model is partitioned into stages (for example, blocks 1–N on one group of devices, blocks N+1–2N on another) and micro-batches are streamed through the pipeline.

Frontier systems combine these strategies with memory-saving tricks such as activation checkpointing and with reduced-precision arithmetic (for example, `float16` or `bfloat16`) to keep training feasible.

Order-of-Magnitude Numbers

Exact figures for frontier-class models are often proprietary, but it is reasonable to expect:

- parameter counts in the range of tens to hundreds of billions,
- training corpora containing trillions of tokens across many languages and modalities,
- and training runs that consume the equivalent of thousands to tens of thousands of accelerator-years of compute.

Remembering these orders of magnitude helps calibrate expectations: the architectures you experiment with on a laptop are qualitatively similar, just scaled down by several powers of ten.

8 Inference, Tools, and Deployment

Once a model is trained, most of the real-world engineering effort shifts to inference: taking a prompt, producing tokens quickly and cheaply, and wiring the model into a product. This

section connects three ideas you will often see bundled together in modern systems: decoding, tool use, and retrieval.

Decoding is the process of turning next-token probabilities into a concrete output sequence. Greedy decoding is simplest but brittle; sampling methods (temperature, top-k, nucleus/top-p sampling) trade determinism for diversity and are often used for creative tasks. In most production settings, latency dominates the user experience, so serving stacks rely heavily on caching: key-value (KV) caches reuse attention computations for previously generated tokens, cutting per-token work during long generations.

Two inference-time extensions have become especially important. The first is *retrieval-augmented generation* (RAG), where the model is conditioned on relevant external documents fetched from a search index [12]. The second is *tool use*, where the model plans and issues structured actions (search queries, calculator calls, database reads, code execution), then incorporates the results into its response [13]. Both aim to improve factuality and capability without retraining the base model.

Finally, speculative decoding accelerates generation by letting a smaller "draft" model propose multiple tokens that a larger model then verifies [14]. Conceptually, it is another instance of the workbook's recurring theme: you change what happens inside the computation graph at inference time, and you get different latency–quality trade-offs at the surface.

Practical Deployment Trade-offs

When you evaluate an LLM deployment, keep three questions in view:

- *Quality:* which decoding strategy, tools, and retrieval sources best fit the task and risk profile?
- *Latency and cost:* how much compute is spent per generated token, and how much can caching or speculative decoding reduce it?
- *Safety:* where do you enforce policy (prompting, tool filters, output filtering, monitoring), and how do you detect failures?

9 Evaluation, Safety, and Governance

Evaluation closes the loop between what we build and what we believe about it. In practice, a responsible workflow combines quantitative benchmarks, qualitative probing, and continuous monitoring after deployment.

Benchmarks offer standardised comparisons, but they are also easy to overfit. Modern evaluation suites therefore mix broad task batteries (for example, BIG-bench [16]) with structured, scenario-based reporting (for example, HELM [15]). For high-stakes settings, teams add red-teaming: adversarial testing designed to surface failure modes that ordinary prompts would not reveal.

Safety and governance are not single techniques; they are layers. Some layers live in training (data curation, instruction tuning, RLHF), others live in inference (tool constraints, retrieval filters, refusals), and many live outside the model (human review, logging, incident response, and policy). A useful mental model is to treat safety as a systems property: it emerges from the interaction between the model, the user, the surrounding tools, and the organisation operating the system.

A Minimal Evals Loop

A simple evaluation loop you can run even on small models looks like this:

- define a small set of tasks that matter (accuracy, helpfulness, refusal behaviour, latency),
- create a fixed prompt suite and a scoring rubric,
- run the suite whenever you change data, decoding, tools, or deployment settings,
- track regressions and investigate with targeted red-team prompts.

This workflow scales up: frontier labs run the same loop, just with larger prompt sets, more automation, and stricter governance.

At this point you have a complete end-to-end picture: how raw inputs become tokens and tensors, how Transformer blocks and attention perform the core computation, how scaling and parallelism make frontier training possible, and how inference-time tooling and evaluation shape what users experience. When you feel lost, return to the concept map in Figure 1. When a term is unfamiliar, use the glossary below; when you want to go deeper, follow the references at the end.

10 Glossary

This glossary provides short definitions and pointers to where terms appear in the workbook and in the references at the end.

Activation checkpointing A memory-saving technique that stores fewer intermediate activations during the forward pass and recomputes them during backpropagation. See Section 7.

Alignment A family of methods and processes that aim to make model behaviour match human intent and constraints, especially in deployment. See Section 9 and [5].

API (application programming interface) A programmatic interface to a model or service (often text-in/text-out, sometimes multimodal), typically exposed over HTTP. See Section 8.

Attention A mechanism that computes a data-dependent weighted average over tokens (or other elements) to mix information across positions. See Section 6 and [1].

Batch size (B) The number of sequences processed together in one forward/backward pass. See Section 4.

Benchmark A standardised evaluation task or suite used to compare models or systems. See Section 9 and [15, 16].

BPE (byte-pair encoding) A tokenisation method that merges frequent symbol pairs to build subword units. See Section 4 and [6].

Causal mask A mask used in autoregressive attention so each token can attend only to itself and earlier tokens. See Section 6.

Context length (L) The number of tokens a model can condition on at once (prompt + generated tokens). See Section 4.

Data parallelism A parallel training strategy where each device holds a full model replica but processes different batches; gradients are aggregated across replicas. See Section 7.

Decoding The method used to turn next-token probabilities into an output sequence (greedy, sampling, beam search). See Section 8.

Embedding A learned mapping from token IDs to vectors, producing the initial representation H_0. See Section 4.

Fine-tuning Updating a pretrained model on a narrower dataset or objective (for example, instruction data) to shape behaviour. See Section 3.

FlashAttention A family of attention implementations that reduce memory traffic and improve speed, especially at long context lengths. See Section 7 and [9].

Forward pass / backward pass The forward computation produces outputs; the backward computation propagates gradients for learning. See Section 5.

GPU / TPU Accelerator hardware commonly used to train and serve large neural networks (graphics processing unit; tensor processing unit). See Section 7.

Gradient The derivative of a loss with respect to parameters; used by an optimiser to update weights. See Section 5.

KV cache (key-value cache) Cached attention keys and values from previous tokens that reduce per-token work during generation. See Section 8.

LayerNorm A normalisation operation applied across features that stabilises training. See Section 5.

LLM (large language model) A neural model trained on large text corpora to predict or generate language; modern systems often extend this with tools and multimodal inputs. See Section 1 and [2].

Loss A scalar objective (for example, negative log-likelihood) that training aims to minimise. See Section 3.

MLP / FFN (feed-forward network) The position-wise nonlinear sub-layer inside a Transformer block, often expanding and then projecting back to d_{model}. See Section 5.

Mixed precision Training or inference using lower-precision formats (for example, FP16 or BF16) to improve speed and reduce memory. See Section 7 and [8].

Mixture-of-experts (MoE) A model design where only a subset of expert sub-networks is activated per token, increasing capacity with limited compute. See Section 1 and [11].

Multimodality Conditioning a model on multiple input types (text, images, audio, code) and producing outputs that may also be multimodal. See Section 3.

Optimiser The algorithm that updates parameters using gradients (for example, variants of stochastic gradient descent). See Section 7.

Overfitting When performance improves on a benchmark or training set but fails to generalise to new data or settings. See Section 9.

Parameter count The number of learned weights in a model, a rough proxy for capacity and compute requirements. See Section 7.

Pipeline parallelism Splitting a model into sequential stages across devices and streaming micro-batches through the stages. See Section 7.

Positional encoding A method to inject token order information into the model (learned or fixed). See Section 4 and [1].

Pretraining Training a model on a broad next-token prediction objective before task-specific adaptation. See Section 3 and [2].

Q/K/V (queries/keys/values) The three projected representations used in attention to compute weights and produce a weighted sum. See Section 6 and [1].

RAG (retrieval-augmented generation) Conditioning generation on externally retrieved documents to improve factuality or domain coverage. See Section 8 and [12].

Red-teaming Adversarial testing intended to discover safety and robustness failures. See Section 9.

Residual connection Adding a sub-layer's input to its output, improving gradient flow and stability. See Section 5.

RLHF Reinforcement learning from human feedback, an alignment approach that uses preference data (and often a reward model) to steer behaviour. See Section 3 and [5].

Scaling laws Empirical relationships linking model/data/compute scale to loss and downstream performance. See Section 3 and [3, 4].

SentencePiece A tokenisation framework that learns subword units directly from raw text, often used in multilingual settings. See Section 4 and [7].

Speculative decoding A speedup technique where a smaller model proposes tokens and a larger model verifies them. See Section 8 and [14].

Tensor parallelism Splitting weight matrices across devices so a single layer is computed collaboratively. See Section 7.

Tokeniser The algorithm that converts raw text into a sequence of integer token IDs. See Section 4.

Tool use Letting a model call external tools (search, calculator, code, databases) and integrate their outputs. See Section 8 and [13].

Top-k / top-p sampling Sampling schemes that restrict candidate next tokens to a small set (top-k) or a probability mass threshold (top-p). See Section 8.

Transformer The attention-based architecture underlying most modern LLMs. See Section 5 and [1].

ZeRO A family of optimiser-state and gradient sharding techniques that enable very large models by reducing memory redundancy. See Section 7 and [10].

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention Is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. https://arxiv.org/abs/1706.03762.

[2] T. B. Brown et al. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. https://arxiv.org/abs/2005.14165.

[3] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling Laws for Neural Language Models. 2020. https://arxiv.org/abs/2001.08361.

[4] J. Hoffmann et al. Training Compute-Optimal Large Language Models. 2022. https://arxiv.org/abs/2203.15556.

[5] L. Ouyang et al. Training language models to follow instructions with human feedback. 2022. https://arxiv.org/abs/2203.02155.

[6] R. Sennrich, B. Haddow, and A. Birch. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of ACL*, 2016. https://arxiv.org/abs/1508.07909.

[7] T. Kudo and J. Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of EMNLP*, 2018. https://arxiv.org/abs/1808.06226.

[8] P. Micikevicius et al. Mixed Precision Training. In *International Conference on Learning Representations (ICLR)*, 2018. https://arxiv.org/abs/1710.03740.

[9] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. 2022. https://arxiv.org/abs/2205.14135.

[10] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *Proceedings of SC*, 2020. https://arxiv.org/abs/1910.02054.

[11] W. Fedus, B. Zoph, and N. Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. 2021. https://arxiv.org/abs/2101.03961.

[12] P. Lewis et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. https://arxiv.org/abs/2005.11401.

[13] S. Yao, J. Zhao, D. Yu, N. D. Goodman, and K. R. Narayanan. ReAct: Synergizing Reasoning and Acting in Language Models. 2022. https://arxiv.org/abs/2210.03629.

[14] Y. Leviathan, M. Kalman, and Y. Matias. Fast Inference from Transformers via Speculative Decoding. 2023. https://arxiv.org/abs/2211.17192.

[15] P. Liang et al. Holistic Evaluation of Language Models. 2022. https://arxiv.org/abs/2211.09110.

[16] A. Srivastava et al. Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models. 2022. https://arxiv.org/abs/2206.04615.