Cheng Yin

Large Language and Generative Foundation Models

Foundations, Systems, Alignment, and Applications

June 14, 2026

Springer Nature

Preface

This book is written as a modern, research-grounded introduction to large language models. The prose, organization, notation, examples, and arguments are original. Each chapter is intended to connect three views that are too often taught separately: the mathematical object being optimized, the system that makes the optimization feasible, and the evaluation procedure that determines whether the result is useful.

The book deliberately avoids an implementation-by-implementation narrative. Instead, it follows the lifecycle of a model: data and tokenization, architecture, pretraining, distributed systems, serving, adaptation, alignment, retrieval, reasoning, multimodality, and safety. Historical systems such as GPT, Alpaca, LoRA, RLHF, and LLaMA are treated as conceptual anchors rather than as recipes to copy without context. The intended reader should come away able to ask what is being optimized, what evidence supports the claim, what system constraint changes the result, and what evaluation would make a deployment decision credible.

This is also not a catalog of model names. Model families change quickly, but the contracts behind them are more durable: tokenization defines the interface, pretraining defines the base distribution, post-training defines assistant behavior, inference systems define usable latency and cost, and governance defines what evidence is enough to ship. The chapters therefore emphasize invariants, accounting identities, failure modes, and reproducible records over leaderboard snapshots.

The exposition is evidence-first. Claims about algorithms, scaling behavior, evaluation, or deployment are tied to primary papers or technical reports whenever possible. When a method is still moving, the text separates the stable idea from the particular implementation that made it visible. No third-party slide text, media transcript, dataset record, model output, or code listing is reproduced as manuscript prose. The local course repository and surrounding materials are used as a coverage map and source of exercise ideas, not as text to copy.

Readers can use the book in three passes. A first pass should follow the chapter summaries, key terms, and exercises to build a conceptual map. A second pass should work through the equations, tables, and system diagrams as implementation contracts. A third pass should use the citations, appendix run-card template, and evaluation cautions to design reproducible experiments or review a model report critically.

Declarations, Provenance, and Responsible Use

This manuscript is prepared from original explanatory text and public research papers or technical reports. It does not reproduce third-party slides, videos, dataset records, model outputs, code listings, or figures. When a public software project, dataset, model report, benchmark, law, or policy document is discussed, it is cited as a source rather than copied as content.

The book discusses model training, deployment, alignment, data collection, evaluation, retrieval, and agentic systems. These topics carry risks involving privacy leakage, copyright infringement, unsafe generation, benchmark contamination, dual-use automation, and misleading evaluation. Chapters therefore include explicit safety, licensing, and evaluation cautions where those risks are material.

Readers who reproduce examples should treat data provenance, license compatibility, personally identifiable information, and benchmark contamination as experimental variables rather than administrative afterthoughts. A run that trains successfully is not automatically publishable. A model that answers a benchmark well is not automatically reliable under distribution shift. A retrieval or tool system that works in a notebook is not automatically safe when connected to private documents, external APIs, or actions that affect other people.

The manuscript also separates educational mechanisms from deployment recommendations. Simplified objectives, toy datasets, small models, and controlled failure cases are used to make technical ideas inspectable. They should not be read as sufficient governance, privacy, security, or safety controls for production systems.

No claim in the manuscript should be interpreted as legal advice, medical advice, security advice, or a recommendation to deploy an unreviewed model in a high-stakes environment. Deployment decisions require independent review of data rights, domain risk, evaluation coverage, monitoring, incident response, and applicable law.

Contents

Acronyms

API	Application programming interface.
BPE	Byte-pair encoding.
CoT	Chain-of-thought prompting.
DPO	Direct preference optimization.
FFN	Feed-forward network.
FSDP	Fully sharded data parallelism.
GQA	Grouped-query attention.
KV	Key-value.
LLM	Large language model.
LoRA	Low-rank adaptation.
MoE	Mixture of experts.
MQA	Multi-query attention.
PEFT	Parameter-efficient fine-tuning.
PPO	Proximal policy optimization.
RAG	Retrieval-augmented generation.
RLHF	Reinforcement learning from human feedback.
RLVR	Reinforcement learning with verifiable rewards.
RoPE	Rotary position embedding.
SFT	Supervised fine-tuning.
VLM	Vision-language model.

Part I
Foundations

Chapter 1
What Makes a Language Model Large

Abstract Large language models are not defined by parameter count alone. They are coupled artifacts whose behavior emerges from data selection, tokenization, architecture, optimization, post-training, inference systems, and evaluation. This chapter establishes that lifecycle view. It explains why causal prediction became the dominant training interface, why scaling laws changed the engineering economics of language modeling, why instruction following and preference learning turned pretrained models into assistant systems, and why reasoning models made inference-time computation part of the modeling problem.

Chapter contract.

The reader should leave this chapter able to describe a large language model as a lifecycle stack rather than a parameter count, identify the major axes of scale, explain why next-token prediction became a general interface, and distinguish release evidence from model-name marketing.

1.1 The Object of Study

A language model assigns probabilities to token sequences, but a modern large language model is better understood as a trained conditional computation system whose distribution is shaped by corpus design, optimizer dynamics, architectural constraints, and deployment-time decoding. For a tokenized sequence $x_{1:T}$, the causal language-model factorization is

$$p_\theta(x_{1:T}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}), \tag{1.1}$$

and the usual training objective minimizes the average negative log-likelihood,

Table 1.1 Axes of largeness in modern language-model systems.

Axis	Symbol	What Scales	Typical Failure Mode
Parameters	N	Dense weights or total MoE weights	Capacity improves while memory, bandwidth, and overfitting risks grow
Active compute	N_{active}	Parameters used per token in sparse models	Routing imbalance or expert underuse hides nominal scale
Training data	D	Tokens, documents, domains, and languages	More tokens amplify duplication, leakage, and licensing problems
Training compute	C	FLOPs and accelerator time	Undertraining or wasted compute from poor data/model balance
Context	T	Prompt and generated sequence length	KV-cache memory and attention cost dominate serving
Inference budget	B	Samples, search, retrieval, tools, verifiers	Higher pass rates can hide weak calibration or invalid verifiers

$$\mathcal{L}(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t}), \tag{1.2}$$

with perplexity defined as $\exp(\mathcal{L})$ when the loss is measured in natural-log units. The Transformer made this system practical at scale because self-attention exposes long-range token interactions while remaining highly parallel on accelerators [168]. GPT-style causal models then showed that a single left-to-right objective can support broad transfer when the model is trained on enough heterogeneous text [139, 15]. BERT-style masked models remain important for representation learning and retrieval, and dense retrieval systems later made representation quality central to open-domain question answering and RAG pipelines [36, 70, 85].

The word "large" therefore refers to several coupled axes, summarized in Table 1.1. Parameter count changes capacity, but active parameters matter separately in sparse mixture-of-experts systems; token count changes the statistical evidence seen during training, but token provenance determines whether the evidence is useful or publishable; context length changes what can be conditioned on, but KV-cache memory and attention cost determine whether that context can be served economically [68, 56, 34]. Inference budget is now also part of model behavior, because retrieval, reranking, sampling, verification, tool calls, and explicit reasoning traces can spend additional computation after pretraining has ended [13, 85, 24, 187].

This book treats a large language model as a stack rather than as a single neural network, with Figure 1.1 showing the lifecycle view used throughout the book. The stack begins with raw data and a tokenizer; passes through architecture, loss, optimizer, and distributed training; continues through supervised instruction tuning and preference optimization; and ends in an inference service that must decide how much context, retrieval, search, and safety filtering to apply. A mistake in any layer can be misread as a model limitation: a weak tokenizer can harm multilingual coverage, a contaminated benchmark can exaggerate capability, an unstable optimizer can waste compute,

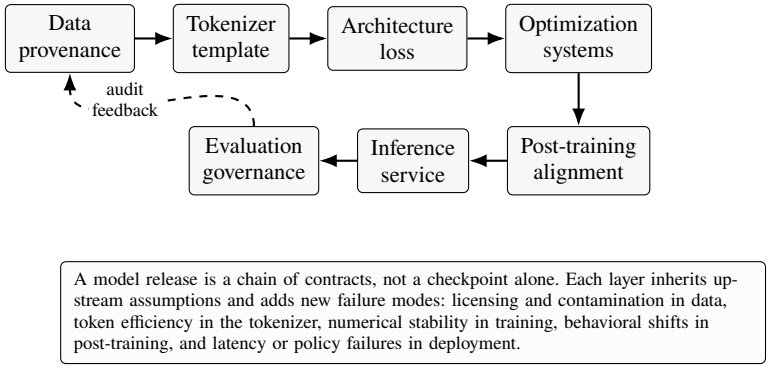

Fig. 1.1 The lifecycle stack used throughout the book. Each arrow represents a contract: the next layer inherits the assumptions, artifacts, and failure modes of the previous layer. The dashed loop emphasizes that evaluation and deployment incidents should feed back into data, training, and release practice rather than remain a final report.

and a poorly designed serving path can make a strong model unusable under latency constraints [153, 180].

1.2 Why Prediction Became a General Interface

The central technical bargain of language modeling is that next-token prediction converts unstructured text into a dense self-supervised training signal. Every token in a document can contribute a target, so web-scale corpora become usable without manual labels [15]. The bargain is imperfect: the objective rewards distributional continuation, not truth, helpfulness, calibration, or harmlessness [125]. Yet it is powerful because language co-occurs with facts, code, dialogue, procedures, arguments, and instructions, so a model trained to predict text can internalize many latent tasks before any task-specific fine-tuning [15].

The Transformer architecture amplified this bargain. Self-attention gives each position a data-dependent path to previous positions, so the model can condition on both local syntax and distant context [168]. Multi-head attention gives the model multiple learned projection subspaces for comparing and aggregating tokens, while feed-forward sublayers supply high-capacity tokenwise transformations. Residual pathways and normalization make very deep stacks trainable, and modern decoder families refine these ingredients with rotary position embeddings, RMS normalization, gated feed-forward networks, grouped-query attention, and mixture-of-experts routing [159, 165, 1, 34].

Prediction also gives a clean implementation path. The model receives token ids, maps them to embeddings, repeatedly mixes information through attention and feed-forward blocks, and projects the final hidden states to vocabulary logits. Training minimizes cross-entropy between predicted logits and the next token, while generation samples or selects tokens from the resulting distribution. Later chapters derive this

computation from tensor shapes and turn the derivations into original implementation exercises.

1.3 Scaling Changed the Research Program

Before the modern scaling era, natural language processing was organized around task-specific datasets, architectures, and losses. Scaling-law work reframed the field by showing that loss follows empirical power laws over model size, dataset size, and compute across broad ranges [68]. Compute-optimal training then shifted attention from simply increasing parameters to balancing model size and token count, arguing that many early large models were undertrained relative to their compute budgets [56]. The consequence is practical: a serious training plan must budget parameters, tokens, sequence length, hardware throughput, and data quality together.

Emergent abilities are the cautionary companion to smooth loss scaling. Wei et al. define an ability as emergent when it is absent in smaller models but present in larger models, so the observed downstream curve looks threshold-like rather than smoothly extrapolated from small-scale behavior [174]. This does not mean the threshold is a universal constant. The apparent onset can move with data quality, model family, prompt format, finetuning, inference method, and metric. Exact-match or accuracy metrics can hide gradual improvements in log probability until a task score suddenly changes. A credible emergence claim should therefore report the full model series, training compute or parameter scale, data and prompt protocol, random baseline, metric choice, uncertainty, and whether smoother quantities such as loss or calibrated likelihood changed before the headline score.

Open model reports made this systems view explicit, although they should be read as technical reports rather than as a substitute for independent replication. LLaMA demonstrated that carefully trained smaller models can be competitive when token budgets and data mixtures are chosen well [165]. Llama 2 expanded the public discussion of pretraining, supervised fine-tuning, preference data, and safety evaluation [166]. Llama 3 pushed the recipe toward longer context, multilingual coverage, coding, safety models, and extensive post-training [48]. DeepSeek-V3 and Kimi K2 indicate that frontier-scale open-weight systems increasingly combine mixture-of-experts routing, specialized attention mechanisms, and agentic or coding-oriented post-training, but their claims still require careful benchmark and deployment-context interpretation [34, 73].

Scaling also exposed failure modes. Training loss can improve while benchmark validity degrades through leakage, contamination, or overfitting to public leaderboards [153, 180]. Larger context windows can increase cost faster than they increase robust long-context reasoning. More samples at inference time can improve pass rates while hiding weak calibration or brittle verification. This book therefore treats evaluation as a modeling component, not as an afterthought.

1.4 From Base Models to Assistants

A pretrained base model is not yet an assistant. It has learned broad continuation behavior, but user-facing behavior requires additional training signals that specify what counts as a helpful response. Supervised instruction tuning supplies demonstrations of desired behavior, and RLHF adds preference rankings so the model can be optimized toward responses humans prefer under a specific annotation protocol [125]. Direct preference optimization and related methods simplify this stage by fitting preferences without an explicit online reinforcement-learning loop, although they still depend on the quality and scope of the preference data [140].

Parameter-efficient adaptation changed who can perform this second stage. LoRA freezes the base model and learns low-rank updates inside selected linear layers, which makes adaptation far cheaper than full fine-tuning for many settings [61]. QLoRA combines low-rank adaptation with quantized base weights, making instruction tuning feasible on much smaller hardware budgets [35]. These methods are not magic compression; they change the trainable subspace, memory footprint, optimizer state, and failure modes. Chapter 10 treats rank, target modules, quantization error, and evaluation as design decisions rather than as fixed defaults.

Assistants also need retrieval and tools when the answer depends on fresh, private, or long-tail knowledge. Retrieval-augmented generation inserts external evidence into the context, while tool-use patterns let the model call search, calculators, databases, or code execution environments [85, 189]. These systems trade parametric memory for runtime grounding, but they introduce their own failure modes: retriever recall, reranker precision, context ordering, citation faithfulness, and prompt-injection risk.

1.5 Reasoning Makes Inference Part of Training

Reasoning in large language models is not a single mechanism. Chain-of-thought prompting elicits intermediate natural-language steps, self-consistency samples multiple paths, tree search branches over candidate partial solutions, and verifier-guided methods score or filter reasoning traces [175, 188]. These traces can improve task performance without being faithful causal explanations of the model's internal computation, so they must be evaluated as generated artifacts rather than accepted as transparent reasoning records [105]. Reasoning-oriented reinforcement learning adds another layer by rewarding correct final answers, verified intermediate steps, or task-specific executable outcomes [33, 176]. A recurring post-2024 trend is that inference-time compute and training-time incentives must be studied together: some gains come from better policies, while others come from searching, sampling, verifying, or allocating more computation at test time [24, 155, 187].

This perspective changes how we teach algorithms such as PPO, DPO, GRPO-style training, process reward models, and Monte Carlo tree search. The book does not present reinforcement learning as a detached prerequisite followed by an unrelated RLHF recipe. Instead, it follows the sequence-level control problem: what is being

rewarded, what distribution is being constrained, what verifier is trusted, and how much extra computation is spent at inference time.

1.6 A Roadmap for the Book

Part I defines the foundation: what is being modeled, how text becomes tokens, how corpora become training signals, and why provenance matters. Part II builds the architecture and systems spine, moving from attention and GPT-style decoders to LLaMA-class blocks, optimization, distributed training, and inference serving. Part III studies adaptation through instruction tuning, synthetic data, LoRA, QLoRA, domain adaptation, and multilingual specialization. Part IV covers retrieval, tools, agents, preference learning, reasoning, multimodality, evaluation, safety, and governance.

The intended reading discipline is simple. Each chapter starts from a concrete problem, states the mathematical object being optimized, connects the object to implementation, and then asks how the claim would be evaluated. That discipline is the main defense against treating slides, benchmark tables, or model cards as self-evident truth. The goal is not to memorize the names of model families; it is to understand why a modeling choice works, when it fails, and how to test it in a reproducible system.

1.7 A Publication-Grade Reading Protocol

A high-quality model book should teach readers how to interrogate claims. For any new model report, read the artifact as a tuple

$$\mathcal{R} = (D, \tau, A, O, S, P, I, E, G), \tag{1.3}$$

where D is the data mixture, τ the tokenizer and templates, A the architecture, O the optimization recipe, S the training system, P the post-training process, I the inference system, E the evaluation evidence, and G the governance constraints. A claim about capability is under-specified if any component is missing. For example, a long-context score without I says little about KV-cache cost or scheduler feasibility; a reasoning score without P and I hides whether the gain came from reinforcement learning, longer traces, more samples, or a stronger verifier.

This protocol also explains why figures in this book are original synthesis diagrams rather than copied architecture art from papers. The purpose of a textbook figure is not to reproduce a model card screenshot; it is to expose the invariant structure that lets readers compare GPT decoders, MoE systems, RAG pipelines, multimodal connectors, and reasoning-time controllers under a single analytic vocabulary. When a diagram is inspired by a paper or system report, the caption cites the source; the drawing itself is a new explanatory abstraction.

1.8 Key Terms

Base model	A pretrained model before instruction tuning, preference optimization, or task-specific adaptation.
Context length	The number of tokens available to condition the model during a forward pass.
Inference budget	The computation spent after a prompt is received, including decoding, retrieval, search, verification, and tool use.
Perplexity	The exponential of average negative log-likelihood under a language model, useful for comparing models only when tokenization and evaluation data are controlled.
Post-training	Training after base pretraining, including SFT, preference optimization, safety tuning, and domain adaptation.

1.9 Exercises

1. Write the causal factorization and cross-entropy loss for a batch with shape $B \times T$. State which tokens should be masked for a chat-style SFT example.
2. Pick two rows from Table 1.1. For each row, describe one benchmark result that could improve while the real system becomes worse.
3. Explain why a model with fewer total parameters can be more expensive to serve than a larger model under a different context length or KV-cache layout.
4. Design a provenance checklist for one dataset, one code repository, one figure, and one model checkpoint before they can be used in this book.
5. Give an example where a chain-of-thought answer is correct but the explanation is not a faithful record of the underlying computation.

Chapter 2
Tokens, Corpora, and Training Signals

Abstract This chapter turns raw text into a defensible training signal. It covers tokenization, corpus construction, deduplication, filtering, contamination control, packing, licensing, and the difference between pretraining, instruction, preference, retrieval, and evaluation data. The central message is that data is not an inert input. It fixes the model's vocabulary, compute budget, benchmark validity, legal exposure, and much of its downstream behavior.

Chapter contract.

The reader should leave this chapter able to audit the data contract behind a model run: what tokenizer is used, what evidence enters the corpus, how examples are packed, which training signal is optimized, and how contamination, licensing, and provenance affect later claims.

2.1 Data as Specification

The first trainable layer of a language model sees integers, not text. Everything before that point is a data contract: which documents are admitted, how they are normalized, how they are split into tokens, how examples are packed into sequences, which targets are masked, and which records are reserved for evaluation. A model trained on noisy, duplicated, or legally unusable data can still reduce pretraining loss, but the resulting artifact may be less useful, less trustworthy, and less publishable than a smaller model trained on a better specified corpus.

Let $\mathcal{D}$ be a collection of documents and let τ be a tokenizer mapping text strings into token ids. A causal pretraining example is usually a token sequence

$$x_{1:T} = \tau(d)_{1:T}, \tag{2.1}$$

and the base training signal is the next-token loss

Table 2.1 Data artifacts in a large language model project.

Artifact	Main Contract	Common Failure Mode
Tokenizer corpus	Defines reusable units and token efficiency	Rare scripts, code, or domain terms become long token sequences
Pretraining corpus	Supplies dense self-supervised targets	Duplication, low-quality text, private data, or benchmark leakage
Instruction data	Demonstrates desired input-output behavior	Labels train on the prompt, formatting leaks, or style dominates task skill
Preference data	Ranks competing responses under a policy	Annotator preferences are underspecified or reward spurious verbosity
Retrieval corpus	Provides evidence at inference time	Chunks are stale, poorly licensed, badly segmented, or not actually retrieved
Evaluation set	Estimates generalization or safety	Contamination, prompt overfitting, weak rubrics, or hidden distribution shift

$$\mathcal{L}_{\text{pre}}(\theta) = -\frac{1}{|\mathcal{B}|} \sum_{(x,t)\in\mathcal{B}} \log p_\theta(x_t \mid x_{<t}). \tag{2.2}$$

The equation hides many decisions. The same document can become a different number of training targets under a different tokenizer; a repeated document can receive many times its intended weight; a benchmark item accidentally included in $\mathcal{D}$ can make a model look more capable than it is; a chat example can train the model to imitate the user if its labels are not masked correctly. Data engineering is therefore part of modeling, not a preprocessing chore.

Table 2.1 separates common data artifacts by the contract they impose. The same raw string should not be reused casually across these roles. Training data teaches behavior, retrieval data grounds behavior at inference time, preference data changes the model's response distribution, and evaluation data measures behavior. Mixing these roles without tracking provenance creates leakage and invalid comparisons.

Figure 2.1 turns that contract view into a data-pipeline manifest that can be audited after training.

2.2 Tokenization as a Modeling Choice

A tokenizer is a lossy modeling interface even when it is exactly reversible at the byte or character level. It chooses the units over which the model predicts probabilities. Word-level tokenizers have intuitive units but fail on rare words, names, spelling variants, code, and multilingual text. Character-level tokenizers avoid unknown words but make sequences much longer. Subword tokenizers occupy the middle: frequent strings become single tokens, while rare strings decompose into smaller pieces.

Byte-pair encoding (BPE) adapts a compression idea to subword segmentation by repeatedly merging frequent adjacent symbols [148]. In a simple word-internal version, the initial vocabulary contains characters, and each merge replaces the most frequent

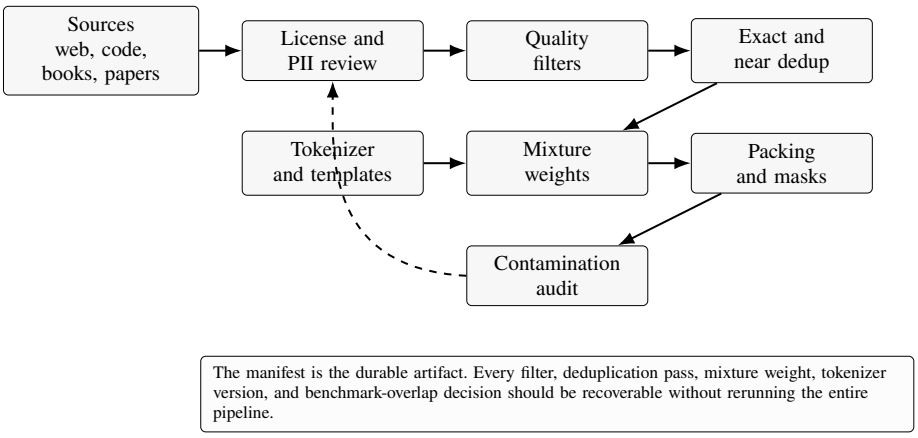

Fig. 2.1 A defensible data pipeline. The diagram is an original synthesis based on data-provenance and contamination concerns in large-model training reports and CS336-style implementation practice.

pair with a new symbol. After M merges, the final vocabulary contains the original alphabet plus learned merges. Modern GPT-style tokenizers often operate close to bytes so they can represent arbitrary Unicode text without an unknown-token failure. SentencePiece made another important design choice: it trains subword models directly from raw text rather than assuming a pre-tokenized word sequence, and it supports both BPE and unigram language-model tokenization [76]. These details matter for languages without whitespace word boundaries, code, mathematical notation, and mixed-script documents.

For a corpus C and tokenizer τ, a useful diagnostic is token efficiency. For language or domain group g, define

$$\rho_g = \frac{\sum_{d \in C_g} \text{bytes}(d)}{\sum_{d \in C_g} |\tau(d)|}. \tag{2.3}$$

Higher ρ_g means more bytes fit into the same context window. This is not automatically better, because bytes are not semantic units, but a very low value often indicates that the model must spend many decoding steps on text that humans would regard as short. Token efficiency is also an inference cost: autoregressive generation pays one forward pass per output token, so a tokenizer that expands a language by 40% relative to another tokenizer imposes a direct latency and cost penalty for that language.

The tokenizer and the model checkpoint are coupled. The embedding matrix has shape $V \times d_{\text{model}}$, where V is vocabulary size. Changing the tokenizer after pretraining changes token ids, special-token conventions, and the distribution of sequence lengths. Vocabulary extension can be useful for domain adaptation, but it is not a harmless metadata edit: new rows in the embedding and output matrices must be initialized and trained, old tokens may split the same string differently, and tied input-output embeddings make the change affect both reading and generation. A production system should therefore version the tokenizer, the chat template, special tokens, normalization rules, and model weights together.

Implementation note.

A tokenizer pipeline should be tested as a reversible function before it is used for training. The tests should include empty strings, whitespace-only strings, non-ASCII names, code indentation, emoji, malformed Unicode, markup, and the exact special tokens used by the model. If a tokenizer has a beginning-of-sequence token, end-of-sequence token, padding token, or chat role markers, the training code should state which component inserts each token. Silent double insertion of BOS or EOS tokens is a common source of off-by-one labels and surprising loss curves.

Educational BPE implementations are useful for learning the algorithm, but their contracts are often not GPT contracts. A word-level BPE script may lowercase text, call a word-punctuation tokenizer, reserve `SOW/EOW/UNK/PAD` markers, and pad every sequence to a fixed length. A GPT-style byte-level tokenizer instead aims to represent arbitrary bytes without an unknown-token path, and training usually packs variable-length text into streams rather than padding every record. A book or model card should therefore say which tokenizer family is being used, how special tokens are inserted, whether text is normalized or lowercased, and whether decoding is expected to be exactly reversible.

2.3 Corpus Construction and Mixtures

A corpus record should be more than a text field. A defensible record includes at least a stable document id, source, acquisition date, license or terms category, language, content type, filtering decisions, and enough lineage to reproduce exclusions. In large projects the raw corpus, filtered corpus, tokenized corpus, and packed training stream should have separate manifests. Without those manifests, it is hard to explain why a model learned a behavior, whether a benchmark was contaminated, or whether a source can be redistributed.

Corpus mixtures are usually heterogeneous. Web pages provide scale and diversity; books provide long-form prose; code provides formal syntax and executable patterns; math and scientific text provide notation and derivations; dialogue and instruction data provide interaction patterns; domain corpora provide specialized vocabulary and tasks. If mixture component i contains D_i available tokens and is sampled with probability p_i, then a training run of S tokens will expose the model to approximately

$$E_i = Sp_i \tag{2.4}$$

tokens from that component, possibly over multiple epochs when $E_i > D_i$. A small high-quality component can therefore be intentionally upsampled, but repeated exposure increases memorization and overfitting risk. Mixture weights are modeling assumptions and should be reported with enough precision to interpret downstream behavior.

Filtering is a sequence of recall-precision tradeoffs. Language identification can remove off-target text but harm code-switching and low-resource languages. Quality classifiers can remove boilerplate but also encode the biases of their training labels.

Toxicity filters can reduce harmful content but remove documents that discuss abuse, medical issues, or newsworthy events in necessary context. Exact rules should be logged because a model trained on the absence of certain topics can fail just as seriously as a model trained on their worst examples.

Deduplication is both a quality intervention and an evaluation intervention. Exact deduplication removes byte-identical or normalized-identical records. Near deduplication uses shingles, hashes, or embedding-like signatures to remove documents with large overlapping spans. Lee et al. show that duplication in language-model datasets can increase memorized generations and distort validation estimates, and that deduplication can reduce memorization while improving training efficiency [81]. The important practical distinction is scope. Deduplicating within the training split reduces overweighted examples; deduplicating between training and evaluation protects benchmark validity; deduplicating across licenses may be required to remove a prohibited source that was mirrored elsewhere.

Implementation note.

Store filtering and deduplication decisions as data, not only as logs. A common pattern is a document table with boolean columns such as `is_language_ok`, `is_quality_ok`, `is_dedup_kept`, and `is_eval_overlap`. That table lets later investigators reconstruct why a document is absent from the final training stream without rerunning every expensive filter.

2.4 Packing Text into Training Sequences

Accelerators prefer dense rectangular tensors, but documents have variable length. A simple batching scheme pads each document to the maximum length in the batch. Padding is easy to reason about, but it wastes compute whenever lengths vary. If a batch has lengths $\ell_1, \ldots, \ell_B$ and maximum length T, the useful-token fraction is

$$\eta_{\text{pad}} = \frac{\sum_{b=1}^{B} \ell_b}{BT}. \tag{2.5}$$

A value of $\eta_{\text{pad}} = 0.55$ means 45% of attention and feed-forward work is spent on padding positions unless kernels explicitly skip them. Pretraining systems therefore usually concatenate tokenized documents into long streams, insert separator or EOS tokens, and slice fixed-length blocks. This approach keeps utilization high but creates another contract: the model may see unrelated documents adjacent in a context window. EOS tokens, document-boundary masks, or careful packing policies determine whether that adjacency becomes a training signal.

For causal language modeling, a packed block of length $T+1$ often yields inputs $x_{1:T}$ and labels $x_{2:T+1}$. This one-token offset is simple, but it is also the source of many bugs. Padding labels should be set to an ignore index, not to the padding token id, unless the

model is intentionally trained to predict padding. In instruction tuning, prompt tokens are commonly masked so that the model is optimized only on the assistant response. If the labels include user tokens, the model is trained partly to imitate the user rather than to answer the user.

Packed datasets also have systems costs. Token streams are large enough that random Python string processing becomes a bottleneck. Practical training pipelines use memory-mapped arrays, binary shards, precomputed indexes, pinned host memory, and asynchronous transfer to the accelerator. Sequence length is a compute multiplier: attention score tensors scale as BHT^2 for batch size B, heads H, and sequence length T. Even when efficient attention kernels reduce memory traffic, longer packed blocks still increase arithmetic and KV-cache size.

Fixed-block binary datasets add one more reproducibility contract. A robust shard records a small header with magic bytes or version, dtype, chunk size, and block size, then stores token ids in fixed-size chunks that can be memory-mapped without reparsing text. Distributed loading must shard files or chunks by both process rank and dataloader worker id; sharding only by worker id can make different data-parallel ranks consume the same examples. If chunks are prefilled with a separator token, shuffled by block index, or combined from weighted datasets, the separator id, random seed, worker cursor, and dataset weights belong in the run manifest and checkpoint.

The binary stream format is itself part of the data contract. Character-level toy corpora and GPT-2 BPE corpora can be saved as `uint16` because their token ids are below 65,536, but a 100k-plus multilingual vocabulary cannot. The preprocessing artifact should therefore store the dtype, tokenizer or `meta.pkl`, vocabulary size, special-token ids, train/validation split rule, and whether an end-of-text token was inserted between documents. Loading a `uint16` stream with the wrong tokenizer can silently produce plausible-shaped tensors with meaningless ids.

2.5 Training Signals Beyond Pretraining

Pretraining supplies broad continuation behavior, but assistant systems need additional signals. Supervised instruction tuning (SFT) trains on demonstrations. A typical record has a prompt, optional context, and a target response. The loss is still cross-entropy, but the label mask changes:

$$\mathcal{L}_{\text{sft}}(\theta) = -\frac{1}{|\mathcal{A}|} \sum_{t \in \mathcal{A}} \log p_\theta(x_t \mid x_{<t}), \tag{2.6}$$

where $\mathcal{A}$ indexes assistant-response tokens. The formatting template is part of the data. Role tags, stop tokens, and tool-call delimiters must match the inference template, or the model is trained under one protocol and served under another.

Preference data changes the signal from a single target to comparisons. A record may contain a prompt, a chosen response, and a rejected response. RLHF trains a reward model and then optimizes a policy against that reward under a regularization constraint [125]; direct preference optimization fits a policy more directly from paired

preferences [140]. In both cases, the data does not reveal a universal human preference. It reveals preferences under a rubric, annotator pool, user population, and model sample distribution. Changing any of those can change the meaning of the same pair.

Verifier data and tool traces are yet another signal. A math verifier may label only final answers, intermediate steps, or executable programs. A tool-use trace may include hidden retrieval calls, database queries, or code execution. These traces should be separated from ordinary assistant text because they teach different behavior and may include private credentials, proprietary schemas, or brittle tool outputs.

Retrieval corpora are not training labels, but they are still data. Dense retrieval systems use embedding models and nearest-neighbor indexes to provide evidence at inference time [70, 85]. Their chunking, metadata, freshness, and access controls affect the generated answer as much as the base model does. A retrieval corpus should be evaluated for recall and attribution, not only for downstream answer quality, because a generator can produce plausible answers even when the retriever failed.

2.6 Contamination, Evaluation, and Provenance

Evaluation requires separation between the evidence used for training and the evidence used for measurement. Contamination can be exact, near duplicate, paraphrastic, or procedural. A benchmark solution may appear in a GitHub repository; a multiple-choice question may appear in a web page with reordered options; a code benchmark may be present in a tutorial; a synthetic instruction dataset may have been generated from benchmark prompts. Public benchmarks are especially vulnerable because model developers can optimize prompts, filters, and post-training data against them. Recent work on benchmark leakage and contamination detection should be read as a warning that a high score is not self-validating [153, 180].

A practical contamination audit should run at several levels: normalized string matching, token n-gram overlap, document-source overlap, and semantic or template-level review for high-value benchmarks. The audit should be asymmetric. It is not enough to ask whether the exact benchmark item appears in training data; one should also ask whether explanations, answer keys, generated variants, or translated versions appear. When a benchmark is used for model selection, its prompts and labels should be access-controlled like test data in any other empirical science.

Provenance and licensing are publication constraints. Notes, slides, notebooks, code repositories, datasets, figures, and model outputs can be useful research artifacts, but they should not be copied into a textbook unless their license and authorship permit it. The safer pattern is to use such material to identify topics and then write original exposition from primary sources. For datasets, provenance review should record the source license, whether redistribution is allowed, whether personal data may be present, whether robots or platform terms restrict collection, and whether generated derivatives can be published.

Quantitative audit lens.

A useful data chapter should give readers quantities they can compute. For mixture component i, track the tuple

$$m_i = (D_i, p_i, E_i, r_i, q_i, c_i), \tag{2.7}$$

where D_i is available token count, p_i sampling probability, $E_i = Sp_i$ expected exposure in a run of S tokens, $r_i = E_i/D_i$ expected repeat rate, q_i a quality score or filter stratum, and c_i a contamination-risk flag. The repeat rate r_i is especially important: a curated mathematical set may deserve upsampling, but high r_i increases memorization and can make validation look better than deployment. This tuple forces data design to be discussed in the same units as optimization and evaluation.

2.7 Key Terms

Byte fallback	A tokenizer design that can represent arbitrary text by falling back to byte-level units instead of producing unknown tokens.
Contamination	Overlap between training or development data and evaluation data that invalidates a generalization estimate.
Deduplication	Removal or downweighting of exact or near-duplicate records within or across data splits.
Mixture weight	The probability with which a data component is sampled during training.
Sequence packing	Concatenating tokenized records into fixed-length training blocks to reduce padding waste.
Tokenizer-model coupling	The dependency between a model checkpoint and the tokenizer, vocabulary, special tokens, and formatting rules used to train it.

2.8 Exercises

1. Train a tiny BPE tokenizer on a small corpus with English prose, Python code, and Chinese text. Report token counts for the same five strings before and after 50, 500, and 5000 merge operations.
2. Given three data components with available sizes $D = (20, 200, 2000)$ million tokens and mixture weights $p = (0.4, 0.3, 0.3)$, compute how many epochs of each component are seen during a 1 billion token run. Which component has the highest memorization risk?
3. Design a label mask for a chat SFT example with system, user, assistant, and tool messages. State exactly which tokens contribute to the loss and why.

4. Implement an exact decontamination check between a toy training set and a toy benchmark using normalized 13-gram overlap. List two cases that this method will miss.
5. For a retrieval corpus, propose metadata fields that would allow answers to cite source, date, license, and access permissions. Explain how missing metadata can create a safety or publication problem.

Part II
Architectures, Optimization, and Systems

Chapter 3
Transformer Mechanics

Abstract This chapter derives the Transformer from tensor shapes. It explains attention, masking, residual streams, normalization, feed-forward layers, positional information, and the implementation invariants needed to debug a model from first principles. The goal is not to memorize a diagram, but to understand the contracts that make deep attention stacks trainable and implementable.

Chapter contract.

The reader should leave this chapter able to trace tensors through a decoder block, state the shape and mask invariant each operation relies on, connect attention and feed-forward math to implementation checks, and diagnose common bugs before interpreting loss curves.

3.1 The Tensor Contract

A Transformer layer is a map from a sequence of hidden vectors to another sequence of hidden vectors. Let a batch of token ids have shape $B \times T$, where B is batch size and T is sequence length. An embedding table maps ids into vectors of width d_{model}, producing

$$X \in \mathbb{R}^{B \times T \times d_{\text{model}}}. \tag{3.1}$$

Every standard Transformer block preserves this outer shape. Attention mixes information across the T positions, and the feed-forward network transforms each position independently, but the residual stream remains $B \times T \times d_{\text{model}}$ from the first block to the final normalization.

The original Transformer combined self-attention, feed-forward sublayers, residual connections, and layer normalization into an architecture that parallelizes over sequence positions far more effectively than recurrent networks [168]. Modern language models adjust many details, but they keep the same central invariant: the model maintains a

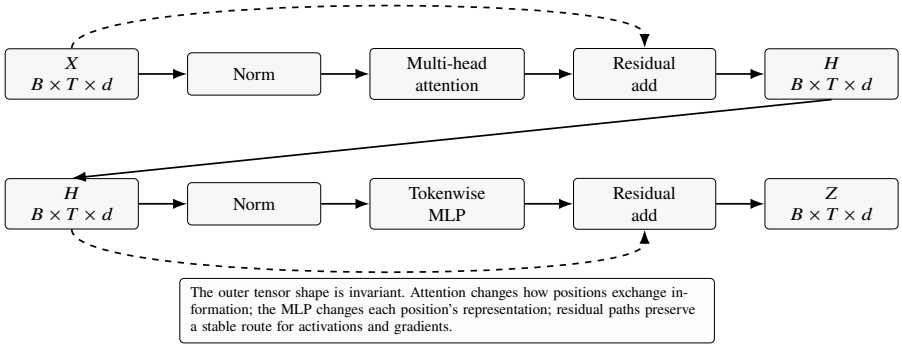

Fig. 3.1 A shape-preserving view of the Transformer block. The attention sublayer changes which positions can exchange information; the MLP changes each position's representation; residual addition keeps a stable path through depth.

residual stream and repeatedly applies content-dependent mixing followed by tokenwise nonlinear transformation.

Figure 3.1 summarizes this shape-preserving contract before the chapter decomposes its parts.

3.2 Embeddings and Position

Token embeddings supply content but not order. If two sequences contain the same token ids in a different order, a permutation-invariant attention layer without position information cannot distinguish them. Position must enter the computation somehow. There are three broad choices.

Learned absolute position embeddings add a trainable vector P_t to each token embedding:

$$X_{b,t} = E[x_{b,t}] + P_t. \tag{3.2}$$

This is simple and was used in GPT-style models, but it fixes a maximum trained context length and gives no built-in rule for unseen positions. Sinusoidal encodings use deterministic sine and cosine functions at multiple frequencies, allowing extrapolation in principle and giving attention a smooth position signal [168]. Rotary position embeddings (RoPE) instead rotate query and key coordinates by a position-dependent angle before the dot product, so relative offsets influence attention scores directly [159].

Position can also enter as an attention bias. ALiBi, for example, adds a head-specific linear penalty as a function of distance, encouraging recency and enabling some train-short, test-long behavior [129]. Bias methods do not require adding position vectors to the residual stream, but they alter the attention score matrix and therefore interact directly with masking and numerical stability.

Implementation note.

Position ids are not always the same as array indices. Packed sequences, left padding, cached decoding, and sliding-window generation can all make the logical position differ from the tensor column. A robust implementation treats position ids as an explicit tensor or constructs them from a well-tested cache update rule.

3.3 Scaled Dot-Product Attention

Self-attention creates three projections of the same residual stream:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V. \tag{3.3}$$

For a single head with head dimension d_h, the attention output is

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}(\frac{QK^\top}{\sqrt{d_h}} + M)V, \tag{3.4}$$

where M is an additive mask or bias broadcastable to the score matrix. The scale factor $\sqrt{d_h}$ keeps dot products in a numerically useful range as the head dimension grows. Without it, the softmax can become overly sharp early in training, reducing gradient signal.

In batched multi-head attention, the projections are reshaped into

$$Q, K, V \in \mathbb{R}^{B\times H\times T\times d_h}, \tag{3.5}$$

where $Hd_h = d_{\text{model}}$. The score tensor has shape $B \times H \times T_q \times T_k$. After softmax and multiplication by V, the head outputs are transposed and concatenated back to $B\times T\times d_{\text{model}}$, then multiplied by an output projection. Multi-head attention is not merely parallel copies of the same computation. Each head has its own learned projections and therefore its own similarity space, value space, mask view, and often its own positional bias.

The computational cost of exact full attention is dominated by the score and value products:

$$O(BHT^2 d_h) = O(BT^2 d_{\text{model}}). \tag{3.6}$$

The memory footprint of materialized attention weights is $O(BHT^2)$. This quadratic term is why context length is a systems decision, not just a modeling hyperparameter.

Numerical stability.

The softmax in Eq. 3.4 should be computed after subtracting the row maximum, and masked positions should be represented by a sufficiently negative additive value before softmax. In mixed precision, many implementations compute score reductions in a

higher precision than the stored activations. A model can appear to have an architecture bug when the real problem is overflow, underflow, or a mask with the wrong dtype.

3.4 Masks and Causality

Masks specify which positions may exchange information. A padding mask prevents real tokens from attending to padded tokens. A causal mask prevents position t from attending to positions $> t$. Encoder-decoder cross-attention uses a different pattern: decoder positions attend to all allowed encoder positions, while decoder self-attention remains causal.

The common causal mask is lower triangular:

$$M_{ij} = \begin{cases} 0, & j \le i, \\ -\infty, & j > i. \end{cases} \tag{3.7}$$

The sign convention matters. An additive mask uses $-\infty$ for disallowed positions; a multiplicative boolean mask may use `True` for allowed positions or for blocked positions depending on the library. Confusing these conventions can train a model that sees the future, attends only to padding, or produces all-`nan` attention rows.

In decoder-only language models, causality is part of the statistical objective. If token x_t can attend to x_{t+1} during training, the model can reduce loss by copying information that will not exist at inference time. This creates a false loss curve and poor generation. Causal masking is therefore not an optional implementation detail; it is the mechanism that makes teacher-forced next-token training match autoregressive decoding.

Teaching encoder-decoder implementations often make the mask convention concrete. A padding mask may be materialized as a Boolean tensor of shape $B \times 1 \times T_q \times T_k$, where allowed query-key pairs are `True`; a decoder self-attention mask is then the logical product of the padding mask and a lower-triangular no-peek mask. This convention matches code that applies `masked_fill` with `mask == 0` and `very_negative` before softmax. It is correct only if every mask uses the same polarity. Cross-attention also has a rectangular $T_{\text{dec}} \times T_{\text{enc}}$ mask, so debugging only square self-attention masks misses an important class of encoder-decoder errors.

3.5 Residual Streams and Normalization

Deep Transformers are trainable because each block modifies a residual stream rather than replacing it. In a pre-normalized decoder block,

$$Y = X + \text{Attention}(\text{Norm}_1(X)), \tag{3.8}$$

$$Z = Y + \text{MLP}(\text{Norm}_2(Y)). \tag{3.9}$$

The residual path gives gradients a direct route through depth, while normalization controls the scale of the inputs to attention and MLP sublayers. Post-normalized blocks instead apply normalization after residual addition. Post-norm was used in the original Transformer, but pre-norm variants are often easier to optimize at substantial depth because the sublayer input distribution is more controlled.

A small translation-oriented reference implementation is therefore not automatically a modern decoder recipe. The original-style stack commonly uses sinusoidal absolute positions, ReLU feed-forward layers, dropout after each sublayer, and the order "sublayer, dropout, residual add, LayerNorm". That is a valid six-layer encoder-decoder teaching model, but a LLaMA-class decoder usually changes several contracts at once: pre-norm RMSNorm, RoPE, gated MLPs, no encoder cross-attention, KV-cache-aware positions, and often a different initialization and optimizer schedule. When porting code, preserve the tensor tests and mask tests, but do not silently inherit the normalization order or positional mechanism.

Layer normalization centers and rescales a vector using its mean and variance. For hidden vector $h \in \mathbb{R}^d$,

$$\mathrm{LayerNorm}(h)_i = \gamma_i \frac{h_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta_i. \tag{3.10}$$

RMSNorm removes the mean-centering step and rescales by the root mean square [195]:

$$\mathrm{RMSNorm}(h)_i = g_i \frac{h_i}{\sqrt{\frac{1}{d}\sum_{j=1}^{d} h_j^2 + \epsilon}}. \tag{3.11}$$

The difference is small in notation but significant in implementation: RMSNorm uses fewer reductions and has become a common default in LLaMA-class decoders.

Normalization does not make scale irrelevant. Residual branches, initialization, optimizer settings, activation functions, and precision still determine whether activations remain in a useful range. A useful debugging habit is to log activation norms per layer during early training. Sudden layer-local norm explosions often reveal a mask, initialization, learning-rate, or mixed-precision error before the loss becomes unrecoverable.

3.6 Feed-Forward Blocks

The feed-forward sublayer is positionwise: it applies the same MLP to each token independently. The original Transformer uses a two-layer network with a larger hidden width:

$$\mathrm{FFN}(h) = W_2 \phi(W_1 h + b_1) + b_2. \tag{3.12}$$

The expansion width is often around $4d_{\text{model}}$ in GPT-style blocks. This sublayer supplies a large fraction of the model's parameters and compute because it is applied at every position in every layer.

Table 3.1 Implementation invariants for a Transformer block.

Check	Expected Property	Symptom When Broken
Residual shape	Every block returns $B \times T \times d_{\text{model}}$	Later projection shape errors or silent broadcasting
Head split	d_{model} divisible by number of heads	Incorrect reshape or mixed head dimensions
Causal mask	Token t attends only to positions $\leq t$	Unrealistically low train loss and bad generation
Padding labels	Padding does not contribute to loss	Model learns to emit padding or loss depends on batch padding
Position ids	Cached and uncached paths use the same logical positions	Prefill works but token-by-token decoding drifts
Softmax rows	Every unmasked query has at least one allowed key	`nan` probabilities after masking

Gated variants split the hidden projection into a value branch and a gate branch. A SwiGLU-style layer can be written

$$\text{SwiGLU}(h) = W_o(\text{SiLU}(W_g h) \odot W_u h), \tag{3.13}$$

where $\odot$ denotes elementwise multiplication. GLU variants have been found effective in Transformer feed-forward sublayers [150]. Modern implementations often shrink the intermediate width relative to the simple $4d$ MLP so the parameter count remains comparable after adding the gate projection.

3.7 Debugging Invariants

Most Transformer bugs are shape bugs, mask bugs, or convention bugs. Table 3.1 lists invariants that should hold before training at scale. These checks are cheap compared with a wasted run.

Attention weights are useful diagnostics but weak explanations. A head that attends to a token does not prove that the token caused the output, because later layers, value projections, residual paths, and MLPs can transform or erase that information. Evaluation of a Transformer should therefore combine mechanistic inspection with controlled interventions, held-out loss, downstream tasks, and generation behavior.

3.8 From Diagram to Tests

The block diagram becomes useful only when it is turned into tests. A high-quality implementation should compare prefill and incremental decoding on the same prefix, assert that logits are invariant under harmless padding, and verify that a single-token continuation uses the same RoPE position as the corresponding full-context computa-

tion. Let $f_\theta^{\text{full}}(x_{1:T})$ be logits from a full forward pass and $f_\theta^{\text{cache}}(x_{1:T})$ be logits from cached token-by-token decoding. Before numerical tolerances from fused kernels are considered, a decoder implementation should satisfy

$$\max_{t \le T} \| f_\theta^{\text{full}}(x_{1:t}) - f_\theta^{\text{cache}}(x_{1:t}) \|_\infty \approx 0. \tag{3.14}$$

When this check fails, the cause is often not attention theory but a position-id, mask, dtype, or cache-layout mismatch. This is why the book emphasizes tensor contracts before architectural folklore.

The training loop has an equally sharp contract. In teacher forcing, the decoder input is the target prefix without its final token, while labels are the same sequence shifted left by one position. The loss mask must ignore target padding, not merely whichever padding id happened to be convenient in the source vocabulary. If source and target tokenizers share the same `<pad>` id in a toy experiment, the bug can stay hidden until the implementation is moved to separate vocabularies, multilingual tokenizers, or chat templates.

3.9 Key Terms

Attention head	One learned query-key-value projection subspace inside multi-head attention.
Causal mask	A mask that prevents a position from attending to future positions.
Residual stream	The shape-preserving hidden state that passes through all blocks by residual addition.
RMSNorm	A normalization layer that rescales by root mean square without subtracting the mean.
Scaled dot-product attention	Attention computed by softmax-normalized query-key dot products divided by $\sqrt{d_h}$.
Tokenwise MLP	A feed-forward network applied independently to each sequence position.

3.10 Exercises

1. For $B = 8$, $T = 1024$, $d_{\text{model}} = 768$, and $H = 12$, write the shapes of Q, K, V, the attention score tensor, the attention output before concatenation, and the final output projection input.
2. Implement single-head causal attention for a batch tensor and verify with an assertion that changing token x_{t+1} cannot change the output at position t.
3. Compare learned absolute positions, sinusoidal positions, RoPE, and ALiBi. For each, state where position enters the computation and one extrapolation risk.

4. Derive the number of parameters in a two-layer MLP with expansion ratio 4 and compare it with a gated MLP that uses two input projections of width m and one output projection.
5. Create a mask convention test for a deep learning library of your choice. Show how a boolean mask and an additive mask represent the same causal pattern.

Chapter 4
From Transformer to GPT

Abstract This chapter specializes the Transformer into a causal decoder. It covers next-token loss, GPT-style blocks, sampling, repetition control, loss curves, minimal training loops, and efficient attention as used in practical GPT implementations. GPT is best understood as an interface: train by teacher-forced next-token prediction, then use the same conditional distribution for autoregressive generation.

Chapter contract.

The reader should leave this chapter able to assemble a causal decoder stack, connect the training loss to logits and sampling behavior, explain why packing and masking choices change the objective, and separate base-model generation from chat-interface contracts.

4.1 The Decoder-Only Contract

A GPT-style model is a stack of masked self-attention blocks trained to predict the next token. The architecture is decoder-only: there is no separate encoder memory, no bidirectional self-attention, and no masked-token objective. For a token sequence $x_{1:T}$, the model defines

$$p_\theta(x_{1:T}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}). \tag{4.1}$$

This factorization turns every position into a supervised target and makes arbitrary text usable as self-supervised data. GPT-2 and GPT-3 demonstrated that the same interface can support broad zero-shot and few-shot behavior when scaled over heterogeneous corpora [139, 15].

GPT-2 made this interface operational by treating naturally occurring web text as latent multitask data rather than as a set of manually labeled task datasets: translation pairs, summaries, question-answer patterns, and reading-comprehension fragments ap-

pear as text. GPT-3 then made the evaluation protocol explicit. Zero-shot, one-shot, and few-shot are inference-time conditioning regimes, not gradient updates. A few-shot result should state the task description, delimiter format, number of demonstrations K, how examples are sampled, context-window budget, answer extraction rule, and whether multiple-choice scoring uses raw, length-normalized, or otherwise calibrated likelihoods.

The training and inference contracts must match. During training, teacher forcing supplies the true prefix $x_{<t}$ for all positions in parallel. During generation, the model receives its own sampled prefix one token at a time. Causal masking is the bridge between the two: it prevents the training computation at position t from seeing targets that will be unavailable during generation.

4.2 Causal Language Modeling

In implementation, a packed block normally contains $T + 1$ token ids. The model input is the first T tokens and the label sequence is shifted by one:

$$\text{input} = x_{1:T}, \qquad \text{label} = x_{2:T+1}. \tag{4.2}$$

If logits have shape $B \times T \times V$, where V is vocabulary size, the token-level cross-entropy loss is

$$\mathcal{L}(\theta) = -\frac{1}{N_{\text{valid}}} \sum_{b=1}^{B} \sum_{t=1}^{T} m_{b,t} \log \frac{\exp z_{b,t,y_{b,t}}}{\sum_{v=1}^{V} \exp z_{b,t,v}}, \tag{4.3}$$

where $m_{b,t}$ is 1 for valid labels and 0 for ignored labels. Ignored labels are needed for padding, prompt masking in instruction tuning, and occasionally for boundary tokens when documents are packed.

Perplexity is $\exp(\mathcal{L})$ when natural logs are used. It is a useful loss-scale diagnostic, but it is not a universal capability metric. Perplexity depends on tokenizer, corpus, context length, and evaluation preprocessing. A character-level model and a BPE model can assign different numbers of targets to the same string; comparing their perplexities without normalization is misleading.

Implementation note.

The most common GPT data bug is an incorrect shift. A quick test is to feed a known sequence such as `[10, 20, 30, 40]` and inspect one batch. The labels should be `[20, 30, 40, next]` for a stream sample or `[20, 30, 40, ignore]` for a bounded sample. If labels equal inputs, the model is being trained as an autoencoder, not a causal language model.

4.3 A Minimal GPT Block Stack

A minimal GPT configuration fixes vocabulary size V, maximum block size $T_{\max}$, number of layers L, number of heads H, hidden width d_{model}, dropout, and whether linear and normalization layers use bias terms. The forward pass has five stages:

1. Map token ids to token embeddings and add or apply position information.
2. Pass the residual stream through L causal decoder blocks.
3. Normalize the final hidden states.
4. Project hidden states to vocabulary logits.
5. Compute cross-entropy loss if labels are supplied.

GPT-style blocks commonly use pre-normalization:

$$h^{(\ell+\frac{1}{2})} = h^{(\ell)} + \mathrm{MHA}(\mathrm{Norm}(h^{(\ell)})), \tag{4.4}$$

$$h^{(\ell+1)} = h^{(\ell+\frac{1}{2})} + \mathrm{MLP}(\mathrm{Norm}(h^{(\ell+\frac{1}{2})})). \tag{4.5}$$

The attention sublayer uses a causal mask; the MLP sublayer is tokenwise; residual additions preserve shape. Weight tying often reuses the token embedding matrix as the output projection matrix, reducing parameters and making input and output token spaces share representations.

A nanoGPT-style implementation makes these contracts explicit in a few hundred lines of code [69]. The model keeps learned token embeddings and learned absolute position embeddings, ties the token embedding matrix to the language-model head, and often pads the vocabulary size to a hardware-friendly multiple even though GPT-2 checkpoints themselves use 50,257 tokens. When importing GPT-2 weights from a library that represents projections as Conv1D-style matrices, several attention and MLP weights must be transposed before assignment. The residual projection weights are also initialized with a smaller standard deviation, approximately $0.02/\sqrt{2L}$, so residual streams do not grow too quickly with depth.

Optimizer grouping is part of the implementation recipe. Weight decay is usually applied to large matrix weights but not to biases or normalization scale parameters. AdamW, learning-rate warmup, cosine decay, gradient clipping, mixed precision, checkpointing, and distributed data parallelism belong to the training system rather than the mathematical block, but they determine whether the block can be trained reliably. Chapter 6 returns to those details.

Checkpoint metadata should include the model configuration, tokenizer identity, optimizer state, training step, random number state when resumability matters, and enough data-stream state to avoid accidentally repeating or skipping large parts of a corpus. A checkpoint that contains only weights is often sufficient for inference, but it is not sufficient for reproducible training.

4.4 Batching, Packing, and Loss Curves

The number of tokens per optimizer step is

$$N_{\text{tok/step}} = B_{\text{micro}} \times T \times G \times W, \tag{4.6}$$

where B_{micro} is micro-batch size, T is sequence length, G is gradient accumulation steps, and W is data-parallel world size. This quantity matters more than batch size alone because it controls gradient noise scale, throughput reporting, and how quickly the model consumes the corpus.

Minimal GPT training code usually stores a tokenized corpus as a large integer stream rather than as a list of examples. For OpenWebText-style reproduction, GPT-2 BPE ids fit in `uint16`, documents are separated by an end-of-text token, and random offsets sample `x = data[i:i+T]` and `y = data[i+1:i+1+T]`. In distributed runs, the intended global tokens per iteration are recovered by multiplying micro-batch size, block size, data-parallel world size, and gradient accumulation steps; each process should divide the accumulated loss before backpropagation and synchronize gradients only on the last micro-step. These details are not bookkeeping trivia: a wrong accumulation scale changes the effective learning rate, and a missing document separator changes the language-modeling distribution.

A healthy pretraining loss curve is smooth only after averaging. Individual steps can spike because a batch contains unusual data, long code spans, repeated tokens, or numerical instability. Validation loss should be estimated over enough batches to reduce sampling noise, and the validation set should be deduplicated from training data. Compute-optimal planning connects model size, token budget, and loss, but a run can still be wasteful if the data stream is low quality or the tokenizer expands important domains excessively [56].

Loss curves must be interpreted with evaluation context. A falling validation loss on web text does not guarantee better instruction following, factuality, long-context use, or safety. Conversely, a small domain model can have worse broad perplexity and better performance on a narrow, well-specified domain. The correct question is not whether loss improved, but whether loss improved on a measurement distribution that matches the intended claim.

4.5 From Logits to Text

At generation step t, the model produces logits $z_t \in \mathbb{R}^V$ for the next token. Greedy decoding selects $\arg\max_v z_{t,v}$. Greedy decoding is deterministic and useful for some constrained tasks, but it can produce repetitive or brittle text. Sampling draws from a probability distribution, usually

$$p(v \mid x_{<t}) = \text{softmax}(\frac{z_{t,v}}{\tau}), \tag{4.7}$$

Table 4.1 Common decoding controls and their main risks.

Control	Intended Effect	Risk
Greedy	Deterministic highest-probability continuation	Repetition and poor exploration
Temperature	Adjust distribution sharpness	Can amplify low-quality tails or collapse diversity
Top-k	Remove low-rank candidates	Fixed k ignores uncertainty variation across steps
Top-p	Keep a dynamic probability nucleus	Sensitive to calibration and tokenization
Repetition penalty	Discourage repeated tokens or spans	Can damage legitimate repetition in code, poetry, and lists
Stop condition	End generation under protocol rules	String-level stops can fail on token-boundary variants

where $\tau > 0$ is temperature. Lower temperature sharpens the distribution; higher temperature flattens it. Temperature is not a creativity knob in isolation. Its effect depends on model calibration, prompt, vocabulary, and any truncation rule.

Top-k sampling restricts the distribution to the k highest-probability tokens before sampling. Nucleus, or top-p, sampling chooses the smallest set of tokens whose cumulative probability exceeds p and samples within that set. Holtzman et al. introduced nucleus sampling as a response to degeneration in open-ended neural text generation [57]. Repetition penalties, presence penalties, frequency penalties, and no-repeat n-gram constraints further modify logits or candidate sets to discourage loops.

A repetition penalty is a decoding-time logit transform, not a training objective. CTRL-style penalized sampling discounts tokens that already appeared in the generated prefix; a penalty of 1 leaves logits unchanged, while larger values discourage reuse [71]. This can break loops under greedy or low-temperature decoding, but it also changes the probability of legitimate repeats such as variable names, quote marks, list bullets, rhymes, legal clauses, and code indentation. A generation report should state the exact transform, whether it is applied to prompt tokens or only generated tokens, its order relative to temperature, top-k, top-p, and logit bias, and examples where it suppresses valid repetition.

Stopping is also part of decoding. Generation may stop at an EOS token, a maximum length, a user-defined delimiter, a tool-call boundary, or a safety policy. A stop string applied after detokenization is not equivalent to a stop token applied during decoding. Stop logic should be tested on partial tokens, whitespace variants, multilingual punctuation, and adversarial prompts that try to hide delimiters.

Table 4.1 summarizes these decoding controls as operational choices rather than decorative sampling knobs.

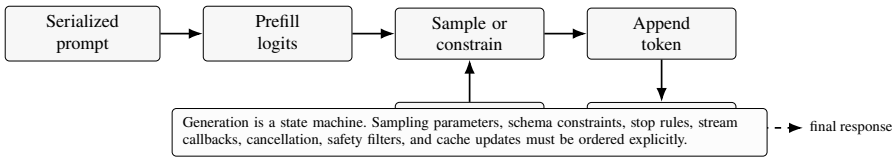

Fig. 4.1 A GPT-style generation loop. The loop is mathematically simple but operationally rich: every token updates model state, user-visible stream state, and service resources.

4.6 Efficient Attention and Generation

Training a GPT block over a length-T sequence computes all positions in parallel. Generation has two phases. In prefill, the prompt is processed as a full sequence and the model creates key-value tensors for every layer. In decode, one new token is processed at a time while reusing the previous keys and values. Without a KV cache, generating n tokens after a prompt would repeatedly recompute the entire prefix. With a KV cache, each step computes new queries, keys, and values for the latest token and attends to cached keys and values.

A teaching generation loop may deliberately skip the KV cache. It crops the growing context to the last `block_size` tokens, runs a full forward pass, keeps only the final-position logits, divides by temperature, optionally applies top-k truncation, samples one token, appends it, and repeats. This is compact and useful for checking model behavior, but it is not a serving benchmark: the loop recomputes the prefix every step and its sampling order becomes part of the experiment contract.

The cache size for a dense multi-head decoder is approximately

$$\text{bytes}_{\text{KV}} = 2 \times B \times L \times T \times H \times d_h \times \text{bytes\_per\_element}. \tag{4.8}$$

The factor of 2 stores keys and values. Long contexts and large batches can make KV memory the limiting serving resource even when model weights fit comfortably on the device.

FlashAttention improves exact attention by changing how the computation is tiled through the memory hierarchy [31]. It does not approximate the softmax attention formula; it avoids materializing large intermediate attention matrices in high-bandwidth memory and recomputes or streams quantities in an IO-aware way. This distinction is important. Some efficient attention methods change model semantics; FlashAttention is an implementation of the same attention operation under suitable masks and precisions.

Efficient generation also depends on batching policy. Continuous batching improves device utilization by admitting new requests as others finish, but it complicates cache layout and stop handling. Prefix caching can reuse KV states for shared prompts, but it requires exact tokenizer and prompt-template equality. Quantized KV caches reduce memory but can affect long-context quality. These are serving-system choices, yet they feed back into model design through context length, number of KV heads, and tokenizer efficiency.

Figure 4.1 makes this generation state machine explicit.

4.7 Evaluation Cautions

GPT evaluation should separate loss, decoding behavior, and task accuracy. Loss is measured under teacher forcing; generation is measured under the model's own sampled prefixes. A model can have lower loss and worse open-ended samples if decoding is poorly chosen. A model can score well on a benchmark and fail in deployment if the benchmark is contaminated, too narrow, or unlike user prompts. A decoding configuration can improve a demo while reducing factuality or calibration.

Report the tokenizer, context length, prompt format, decoding settings, number of samples, random seeds when relevant, and exact metric. For stochastic decoding, a single sample is a weak estimate. For pass-at-k style coding or reasoning metrics, increasing k spends more inference compute and should not be compared directly with a single-sample result unless that is the intended product budget.

4.8 Interface Contracts for Chat Models

Chat models are often described as if the only difference from a base decoder were additional instruction data. Operationally, the difference is a contract between tokenizer, template, loss mask, decoding policy, and runtime. If a conversation is serialized as

$$s = \langle \text{system} \rangle u_0 \langle \text{user} \rangle u_1 \langle \text{assistant} \rangle a_1 \cdots , \tag{4.9}$$

then training must define which spans are labels, which spans are conditioning context, which spans are tool observations, and which token sequences terminate a message. This contract should be tested like an API. A model can lose capability simply because the training template and serving template disagree about a role marker or end-of-message token.

4.9 Key Terms

Autoregressive generation Producing tokens one at a time while conditioning each step on previously generated tokens.

Causal language modeling Training a model to predict each token from only earlier tokens.

In-context learning Specifying a task at inference time with instructions and examples in the prompt rather than by updating model weights.

KV cache Stored key and value tensors reused during decoding to avoid recomputing the prefix.

Nucleus sampling Sampling from the smallest token set whose cumulative probability exceeds a threshold p.

Repetition penalty A decoding-time logit or candidate-set modification that discourages tokens or spans already present in the generated context.

Teacher forcing	Training with true previous tokens supplied as context for every target position.
Weight tying	Sharing the token embedding matrix with the output projection matrix.

4.10 Exercises

1. For a packed token array of length 17 and block size 8, construct two training examples using the $T + 1$ shift convention. Mark inputs, labels, and ignored labels.
2. Implement a tiny GPT block with shape assertions after embedding, attention, MLP, final normalization, and logits. Test that logits have shape $B \times T \times V$ during training and $B \times 1 \times V$ during cached single-token decoding.
3. Train a character-level causal model on a small public-domain text. Plot training loss and validation loss, then explain one sign of overfitting and one sign of a data or label bug.
4. On the same prompt and model, compare greedy, temperature sampling, top-k, and top-p decoding. Hold maximum length and stop conditions fixed, and describe how conclusions change across at least five random seeds.
5. Design zero-shot, one-shot, and few-shot prompt formats for the same classification benchmark. Specify the delimiter, demonstration sampling rule, answer extraction rule, and likelihood normalization used for scoring.
6. Add a repetition penalty to a generation loop. State whether prompt tokens count as repeated tokens, where the processor runs relative to temperature and truncation, and give one example where the penalty harms a valid repeated token.
7. Compute KV-cache memory for a 24-layer model with $B = 8$, $T = 4096$, $H = 16$, $d_h = 64$, and bfloat16 cache tensors. How does the answer change if grouped-query attention uses 4 KV heads?

Chapter 5
LLaMA-Class Architectures

Abstract This chapter studies the architectural conventions used by modern open decoder models: RMSNorm, RoPE and YaRN-style scaling, SwiGLU-style feed-forward networks, grouped-query attention, KV cache design, latent attention variants, long-context extensions, and mixture-of-experts variants. LLaMA-class models are not a different species from GPT; they are a refined decoder-only recipe whose details change stability, memory, throughput, and context behavior.

Chapter contract.

The reader should leave this chapter able to compare LLaMA-class architectural choices against the basic GPT contract from the previous chapter, quantify their training and serving consequences, and ask which evidence is needed before treating a design as an improvement.

5.1 What Makes a Decoder LLaMA-Class

Modern open large language models are mostly decoder-only Transformers, but the defaults have changed since early GPT implementations. LLaMA and its successors popularized a compact recipe: pre-normalized decoder blocks, RMSNorm, rotary position embeddings, bias-free linear layers, SwiGLU-style feed-forward networks, careful tokenizer and data choices, and post-training pipelines that make the base model usable as an assistant [165, 166, 48]. Later families add grouped-query attention, longer context, stronger multilingual and code mixtures, tool-oriented post-training, and sparse mixture-of-experts variants [1, 34, 73].

A LLaMA-class block can be written as

$$y^{(\ell)} = x^{(\ell)} + \mathrm{Attn}(\mathrm{RMSNorm}(x^{(\ell)})), \tag{5.1}$$

$$x^{(\ell+1)} = y^{(\ell)} + \mathrm{SwiGLU}(\mathrm{RMSNorm}(y^{(\ell)})). \tag{5.2}$$

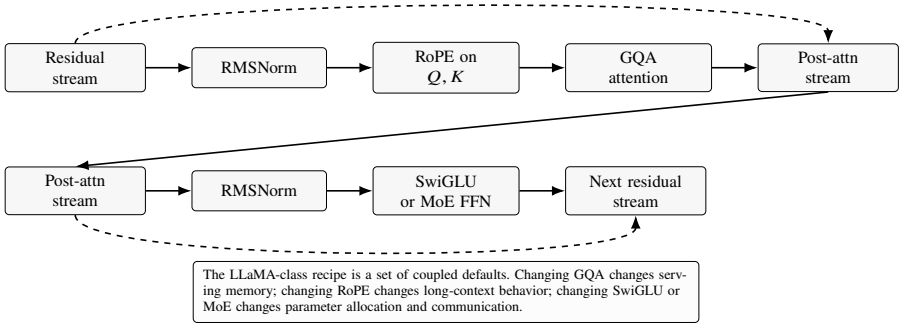

Fig. 5.1 A LLaMA-class decoder block as a contract among normalization, position handling, attention cache structure, and feed-forward capacity. The drawing abstracts from LLaMA, Llama 2/3, DeepSeek-V3, and Kimi K2-style reports without copying any released architecture figure.

This is still the residual-stream structure from Chapter 3. The significance is in the defaults. RMSNorm changes normalization cost and behavior; RoPE changes where positions enter attention; SwiGLU changes MLP expressivity and parameter allocation; grouped-query attention changes the KV-cache budget. These are not cosmetic differences when training or serving billions of parameters.

Figure 5.1 summarizes the coupled defaults that make a decoder LLaMA-class in practice.

5.2 RMSNorm and Pre-Normalization

RMSNorm rescales a hidden vector without subtracting its mean [195]. For $h \in \mathbb{R}^d$,

$$\mathrm{RMSNorm}(h) = g \odot \frac{h}{\sqrt{\frac{1}{d}\sum_{j=1}^{d} h_j^2 + \epsilon}}, \tag{5.3}$$

where g is a learned scale vector. Compared with LayerNorm, RMSNorm removes the mean reduction and bias term in common implementations. The saved work is small per vector but large across every token, layer, and generation request. More importantly, it is part of a block recipe that has been empirically stable for deep decoder-only models.

Pre-normalization means the attention and MLP sublayers see normalized inputs, while residual additions carry unnormalized accumulated state forward. This tends to improve optimization at depth because each sublayer receives a controlled scale even as the residual stream evolves. The final RMSNorm before the language-model head is also important: it makes logits depend on a normalized representation rather than on raw residual scale.

Implementation note.

RMSNorm is easy to implement incorrectly in mixed precision. The mean of squares should usually be accumulated in a sufficiently stable dtype, and ϵ should match the checkpoint family. Changing ϵ or whether the scale is applied before or after casting can produce small logit differences that become visible during long autoregressive generation.

5.3 SwiGLU Feed-Forward Networks

The simple GPT MLP expands from d to $4d$, applies GELU, and projects back to d. LLaMA-class models often use a gated feed-forward network inspired by GLU variants [150]:

$$\mathrm{FFN}(h) = W_o(\mathrm{SiLU}(W_g h) \odot W_u h). \tag{5.4}$$

There are two input projections, W_g and W_u, and one output projection, W_o. To keep parameter count near the ungated $4d$ MLP, the hidden width is often set near $(8/3)d$ and rounded to a hardware-friendly multiple. The gate lets the model modulate candidate features elementwise, while the up projection supplies the candidate values.

The feed-forward block is often the largest parameter component of a dense decoder layer. With hidden width m, the gated MLP has approximately $3dm$ weights, ignoring biases. A standard $4d$ MLP has approximately $8d^2$ weights. Setting $m \approx (8/3)d$ makes these comparable. Rounding m upward can improve tensor-core utilization but changes parameter count, activation memory, and checkpoint size.

5.4 Rotary Position Embeddings

RoPE applies a position-dependent rotation to query and key vectors before the attention dot product [159]. Pair adjacent dimensions and rotate each pair by angle $m\theta_i$ at position m. For a two-coordinate pair u_{2i}, u_{2i+1}, the rotated coordinates are

$$u'_{2i} = u_{2i}\cos(m\theta_i) - u_{2i+1}\sin(m\theta_i), \qquad u'_{2i+1} = u_{2i}\sin(m\theta_i) + u_{2i+1}\cos(m\theta_i). \tag{5.5}$$

If a query at position m and a key at position n are rotated, their dot product contains

$$(R_m q)^\top (R_n k) = q^\top R_m^\top R_n k, \tag{5.6}$$

and $R_m^\top R_n$ depends on the relative offset $n - m$. This is the geometric reason RoPE gives attention access to relative position while remaining compatible with standard dot-product attention.

RoPE is not a free long-context solution. A model trained up to one context length has learned to use a particular range of rotation phases. Extrapolating to much longer contexts can create phase patterns that were rare or absent during training. Long-context

extensions use methods such as position interpolation, frequency scaling, continued training, sliding attention, or attention biases. ALiBi is a contrasting approach that adds linear distance biases rather than rotating query and key vectors [129]. The evaluation question is not whether a model accepts a longer input tensor, but whether it uses distant evidence reliably without damaging shorter-context behavior.

Baichuan 2 is a useful reminder that position handling is an architecture choice, not a family label. Its 7B model uses RoPE while its 13B model uses ALiBi, partly to study both multiplication-based and bias-based attention paths in otherwise related multilingual models [184]. Such a report should not be reduced to "LLaMA-like" or "not LLaMA-like." The positional method affects kernel compatibility, attention masks, long-context extrapolation, and what must be tested during cached decoding.

YaRN-style RoPE extension makes this tradeoff more concrete [127]. Instead of applying one uniform interpolation rule to all rotary dimensions, it treats dimensions by frequency band. The wavelength of dimension i is approximately

$$\lambda_i = \frac{2\pi}{\theta_i}, \tag{5.7}$$

so high-frequency dimensions complete many rotations within the original context and are important for short-range position modeling, while low-frequency dimensions change slowly and carry longer-range information. YaRN preserves high-frequency components, interpolates low-frequency components, blends the middle band, and rescales attention scores so the longer context does not make attention excessively diffuse. A model report that uses YaRN, NTK-aware RoPE, or another scaling rule should state the original training context, target context, scaling factor, frequency-band rule, whether scaling is static or dynamic at inference time, and how short-context regression was tested.

Position interpolation and NTK-aware RoPE are not strict train-free extrapolation. Position interpolation replaces m by m/s, which keeps positions inside a familiar range but compresses high-frequency dimensions as well as low-frequency ones. NTK-aware scaling changes the RoPE base so interpolation strength grows from high to low frequency, preserving short-range phases better than a uniform scale but still changing the query/key distribution. In practice, long-context reports should treat these changes as a continued-training or adaptation recipe unless they show strong no-training evidence across short- and long-context slices.

Implementation note.

RoPE caches must align with the decode cache. In prefill, positions are usually $0, \ldots, T-1$. In cached decoding, the new token's query and key must use the next logical position, not position 0. A one-token error can be almost invisible on short prompts and severe on long generations.

Independent LLaMA implementations make this alignment an API rather than an aside [93]. A common pattern is to precompute a RoPE cache and a lower-triangular mask cache up to `block_size`; during prefill the model slices the first T positions, while

Table 5.1 Attention variants by KV-cache structure.

Variant	KV Heads	Main Benefit	Main Risk
Multi-head attention	H_q	Maximum per-head KV flexibility	Largest decode cache and bandwidth
Multi-query attention	1	Very small KV cache	Quality loss on some tasks or scales
Grouped-query attention	$1 < H_{\text{kv}} < H_q$	Good memory-quality trade-off	Requires compatible checkpoint design
Latent or compressed attention	Model-specific	Smaller or transformed memory state	Harder to compare semantics and kernels

during cached decoding it selects rows by an explicit `input_pos`. If the runtime also supports a fixed `max_seq_length`, then reaching the limit may roll the KV cache left and overwrite the final slot. That is a sliding-window policy, not just an optimization, and it should be reported because it defines which earlier tokens are no longer visible.

Several notebook-scale LLaMA implementations reveal easy mistakes. RoPE is parameterized by head dimension, not model hidden width, and one sine/cosine cache is usually shared across heads and stored as a non-trainable buffer. RoPE is applied to query and key, not value; cached keys should use the same rotated representation that attention reads at decode time. Attention scores should be divided by $\sqrt{d_h}$, not $\sqrt{d_{\text{model}}}$, after heads are split. For GQA, repeating KV heads with `repeat_interleave` is a convenient educational way to show sharing before the dot product, but the serving cache should store only H_{kv} heads. Physically expanding cached keys and values back to H_q heads gives up much of the memory and bandwidth benefit.

5.5 KV Cache and Attention Variants

During autoregressive decoding, every layer stores keys and values from previous tokens. For a dense model with L layers, batch B, context length T, head dimension d_h, and H_{kv} key-value heads, the approximate cache memory is

$$\text{bytes}_{\text{KV}} = 2BLTH_{\text{kv}}d_h s, \tag{5.8}$$

where s is bytes per element. Multi-head attention has $H_{\text{kv}} = H_q$, one KV head per query head. Multi-query attention shares a single KV head across all query heads, reducing cache memory and memory bandwidth during decoding [149]. Grouped-query attention uses an intermediate number of KV heads, grouping several query heads per KV head, and often preserves more quality than full multi-query attention at similar serving benefits [1].

Table 5.1 compares the attention variants by the KV-cache structure they impose.

Equation 5.8 explains why grouped-query attention became a practical default. Weight memory is paid once per loaded model replica, but KV memory grows with active batch and context. In a long-context serving system, KV cache can dominate

capacity. Reducing H_{kv} improves throughput not only by saving memory, but also by reducing memory bandwidth in token-by-token decoding.

Some newer architectures go beyond GQA with latent or compressed attention states, specialized cache layouts, or sparse attention patterns [34]. DeepSeek-V3's multi-head latent attention is a useful example: instead of storing ordinary per-head key and value tensors, the model compresses part of the attention state into low-rank latent variables and then reconstructs the quantities needed for attention. This is closer to changing the cache representation than merely reducing the number of KV heads. It also creates implementation details that must be reported: latent dimension, which components receive RoPE, whether latent states or reconstructed keys are cached, and which kernels support the layout.

An MLA implementation should state exactly what is cached. Caching reconstructed multi-head keys and values gives up most of the memory gain; the intended serving path caches the compressed KV latent and the decoupled RoPE key, then reconstructs or absorbs the remaining projections when attention is computed. The RoPE split is not cosmetic: if the position-sensitive key component is left inside the low-rank reconstruction, every decode step may require recomputing rotated keys for the whole prefix. Matrix absorption, such as folding key-up or value-up projections into the query or output paths after training, is therefore part of the inference contract, not only an algebraic simplification.

These variants should be evaluated under the exact serving regime of interest. A method that improves peak throughput for long prompts may add overhead for short prompts; a cache compression that preserves average benchmark scores may fail on retrieval-heavy tasks requiring exact distant evidence. The comparison should therefore include cache bytes per active token, prefill/decode latency, long-context retrieval accuracy, and whether the implementation depends on custom attention kernels.

5.6 Tokenizer and Vocabulary Contracts

LLaMA-class checkpoints inherit the tokenizer-model coupling from Chapter 2. The vocabulary determines embedding and output projection shapes, special-token ids, chat-template conventions, and token efficiency. Open model families often change tokenizer vocabularies across generations to improve multilingual and code coverage. Such changes are architectural from the checkpoint's point of view: the embedding table, output head, and all learned token statistics are tied to the tokenizer.

Baichuan 2 makes the multilingual version of this contract concrete. It expands the vocabulary to 125,696 tokens, uses SentencePiece BPE without text normalization, splits numbers into individual digits, adds whitespace-only tokens for code, and relies on byte fallback for rare characters [184]. These are not preprocessing trivia. Digit splitting changes arithmetic and numeric copy behavior; whitespace tokens affect code generation; byte fallback defines how rare names and mixed-script text survive tokenization; and a larger vocabulary changes embedding, output-head, and softmax costs.

Vocabulary size is also a systems parameter. Embedding and output matrices scale as Vd_{model}. During training, the final softmax over V can be expensive; during inference, logits processors and sampling operate over V each step. Implementations often pad vocabulary size to a multiple that improves matrix-kernel efficiency. Padding rows must not become ordinary sampled tokens.

Checkpoint conversion is part of this contract. For example, a lightweight implementation may store Q, K, and V as one stacked projection for code simplicity, while a source checkpoint stores separate matrices or multiple model-parallel shards. Converting such weights requires concatenating or reordering tensors along the correct dimension, preserving the tokenizer file, and distinguishing true vocabulary size from padded vocabulary size. If the converted checkpoint can generate text but its parameter accounting, sampling mask, or tensor-parallel merge metadata is wrong, later fine-tuning and serving failures become hard to diagnose.

5.7 Long Context

Long context has three separate meanings. The first is acceptance: the model can run a forward pass at length T. The second is retrieval: the model can identify relevant information somewhere in that context. The third is reasoning or synthesis: the model can combine distant evidence correctly. Many failures occur because a system demonstrates the first and claims the third.

Training for long context increases attention cost, activation memory, data requirements, and evaluation burden. Continuing training at longer sequence lengths can adapt positional behavior, but it may also change short-context performance. Synthetic needle-in-a-haystack tests are useful sanity checks but incomplete. A model can find a unique key phrase and still fail when evidence is paraphrased, conflicting, tabular, or distributed across sections. Long-context evaluation should include retrieval, aggregation, contradiction handling, citation faithfulness, and latency/cost reporting.

Serving long context is dominated by prefill and cache pressure. Prefill cost grows with the prompt length and full attention pattern; decode cost grows with cache length. Sliding-window attention, retrieval-augmented generation, summary memory, and cache eviction policies are alternatives to simply increasing $T_{\max}$. These alternatives change the task semantics and should be exposed in evaluation.

5.8 Mixture-of-Experts Variants

Mixture-of-experts (MoE) models replace some dense feed-forward computation with a set of expert networks and a router. For token hidden state h_t, a router produces scores over experts. A top-k router selects a small subset S_t and computes

$$\text{MoE}(h_t) = \sum_{e \in S_t} \alpha_{t,e} E_e(h_t), \tag{5.9}$$

where E_e is expert e and $\alpha_{t,e}$ is a routing weight. The model can have many total parameters while activating only a fraction per token. This changes the meaning of model size: total parameters, active parameters, routing overhead, and communication cost must be reported separately.

Sparse computation has failure modes that dense models do not. Routers can collapse onto a few experts, leaving others undertrained. Load-balancing losses can improve hardware utilization while interfering with specialization. Expert parallelism introduces all-to-all communication across devices. Capacity limits can drop or reroute tokens. Evaluation can hide these problems if it reports only aggregate accuracy rather than per-domain behavior, routing distribution, latency percentiles, and failure cases.

MoE models such as DeepSeek-V3 and Kimi K2-style systems show how sparse scaling can support strong open-weight performance claims, but the systems contract is more complex than a dense LLaMA-class model [34, 73]. DeepSeek-style MoE designs also illustrate that routing policy is part of the architecture, not only a systems afterthought. Shared experts, routed experts, top-k selection, sequence-level load balancing, expert-parallel placement, and auxiliary or bias-based balance mechanisms all change which tokens update which experts and which devices become bottlenecks.

A publishable comparison should specify active parameters, total parameters, context length, data mixture where known, post-training recipe, inference budget, and serving hardware. For sparse models it should additionally report number of experts, selected experts per token, shared-expert capacity, routing distribution, load-balance objective or bias update rule, token dropping or rerouting policy, and all-to-all communication cost. Without these details, a model-size number can hide both the real compute used per token and the real reason a sparse model is fast or slow.

5.9 Beyond the LLaMA-Class Decoder

Decoder-only Transformers remain the reference architecture for most deployed large language models, but they are not the only plausible sequence backbone. Retentive networks, selective state-space models, and hybrid memory architectures try to change the long-context cost model by replacing all-pairs attention with recurrent or chunkwise state updates [160, 49, 32, 11]. The attraction is clear: if a model can summarize the prefix into a compact recurrent state, decoding no longer needs a KV cache that grows linearly with every past token in every layer.

These architectures should be compared by contract, not by slogan. Attention exposes direct content-addressed access to previous token representations; recurrent or state-space layers compress history into a state whose update rule determines what can be remembered, forgotten, or retrieved. Mamba-style selective state spaces make the dynamics input-dependent, improving content selectivity relative to older SSMs [49]. Mamba-2 and related state-space duality work further connect attention-like computation and structured recurrent algorithms [32]. RetNet proposes parallel training with recurrent inference, while memory-augmented designs such as Titans make test-time memory a learned component rather than a fixed KV table [160, 11].

For a book about large language models, the practical conclusion is conservative. These models are essential frontier material for long-context and streaming systems, but a replacement claim needs evidence at comparable data scale, tokenizer, training budget, context distribution, and serving workload. Many promising sequence models win on asymptotic or medium-scale efficiency but must still prove broad instruction following, retrieval fidelity, tool use, and post-training compatibility at frontier scale.

5.10 Evaluation Cautions

Architecture claims should be tested under matched training and serving budgets. Replacing LayerNorm with RMSNorm, GELU with SwiGLU, MHA with GQA, or absolute positions with RoPE changes parameter count, activation memory, cache memory, and often the optimal learning rate. A fair ablation controls for tokens, compute, data order, tokenizer, optimizer, and evaluation prompts as much as possible.

Open model reports are valuable primary sources, but they are not interchangeable with independent replication. They often combine architecture changes with data improvements, post-training changes, filtering, safety tuning, and benchmark-specific evaluation choices. When a report attributes gains to an architecture, ask whether the evidence isolates that factor. When a deployment chooses a LLaMA-class model, evaluate the actual checkpoint and decoding path rather than relying on the family name.

5.11 Architectural Accounting

For a dense decoder with L layers, hidden width d, attention heads H_q, KV heads H_{kv}, head dimension d_h, and SwiGLU width m, a rough per-layer parameter count is

$$P_{\text{layer}} \approx d(H_q d_h) + 2d(H_{\text{kv}} d_h) + d(H_q d_h) + 3dm, \tag{5.10}$$

where the first four terms correspond to W_Q, W_K, W_V, and W_O, and the last term is the gated feed-forward network. This approximation ignores embeddings, norms, biases, and rounding, but it teaches the key lesson: GQA primarily reduces KV projections and serving cache, whereas the FFN often dominates dense parameters. In an MoE layer, $3dm$ should be replaced by the sum over experts for total parameters and by top-k experts for active parameters. A publishable architecture section should always state which quantity it is reporting.

5.12 Key Terms

Activated parameters	Parameters used for a particular token or forward pass, especially in sparse MoE models.

Grouped-query attention	Attention in which multiple query heads share each key-value head.
KV-cache memory	Memory required to store past keys and values for autoregressive decoding.
Latent attention	An attention design that caches or computes through compressed latent states rather than ordinary full key/value tensors.
Load balancing	Router-side mechanisms that prevent a few MoE experts from receiving too many tokens while other experts remain underused.
RoPE	Rotary position embeddings, which rotate query and key dimensions by position-dependent angles.
RMSNorm	Normalization that rescales activations by their root mean square.
SwiGLU	A gated feed-forward layer using a SiLU gate and elementwise multiplication.
YaRN	A RoPE extension method that preserves high-frequency short-context behavior while interpolating lower-frequency dimensions for longer contexts.

5.13 Exercises

1. Modify a GPT-style block on paper into a LLaMA-style block. List every changed component: normalization, bias terms, position handling, MLP activation, and attention cache assumptions.
2. For $d = 4096$, compare the approximate MLP parameter counts for a GELU MLP with hidden width $4d$ and a SwiGLU MLP with hidden width $(8/3)d$ rounded up to the nearest multiple of 256.
3. Derive the relative-position property of RoPE by multiplying the transpose of R_m by R_n for a two-dimensional coordinate pair.
4. Compare position interpolation, NTK-aware RoPE scaling, and YaRN for a model extended from 32k to 128k context. Which parts of the report would show whether short-context behavior was preserved?
5. Compute KV-cache memory for MHA, MQA, and GQA with $L = 32$, $B = 16$, $T = 8192$, $H_q = 32$, $H_{\text{kv}} \in \{32, 8, 1\}$, $d_h = 128$, and bfloat16 elements.
6. Design an ablation plan to test whether GQA changes downstream quality independently of total parameter count and training tokens.
7. For an MoE layer with 64 experts, top-2 routing, and capacity limits, name three metrics besides benchmark accuracy that should be reported to make the architecture claim credible.
8. Read a DeepSeek-V3-style architecture report and separate architecture claims from systems claims. Which statements depend on MLA, MoE routing, FP8 training, custom kernels, or expert-parallel communication?

Chapter 6
Optimization and Pretraining

Abstract This chapter connects the mathematical objective of language-model pretraining to the engineering recipe used to make large runs stable. It covers autoregressive loss accounting, auxiliary multi-token prediction objectives, AdamW, parameter groups, learning-rate schedules, warmup, gradient clipping, mixed precision, checkpoint recovery, data order, loss prediction, compute-optimal planning, and evaluation cautions.

Chapter contract.

The reader should leave this chapter able to read or design a pretraining run card, relate optimizer and data-order decisions to the objective, budget compute and memory before launching a run, and decide what monitoring evidence is strong enough to stop or continue training.

6.1 The Pretraining Problem

Pretraining is the stage in which a model learns a broad distribution before any assistant-specific behavior is imposed. For a causal decoder, the training example is a token sequence $x_{1:T}$ and the objective is next-token prediction:

$$\mathcal{L}(\theta) = -\frac{1}{M} \sum_{(b,t)\in\mathcal{M}} \log p_\theta(x_{b,t} \mid x_{b,<t}), \tag{6.1}$$

where $\mathcal{M}$ is the set of positions that contribute to loss. In simple packed pretraining, $\mathcal{M}$ contains almost every non-padding position. In instruction tuning, many prompt tokens are masked, but base pretraining usually treats the stream itself as the supervision signal. This difference matters: a pretraining loss curve is a measurement of distributional compression over the corpus mixture, not a direct measurement of helpfulness, truthfulness, or safety.

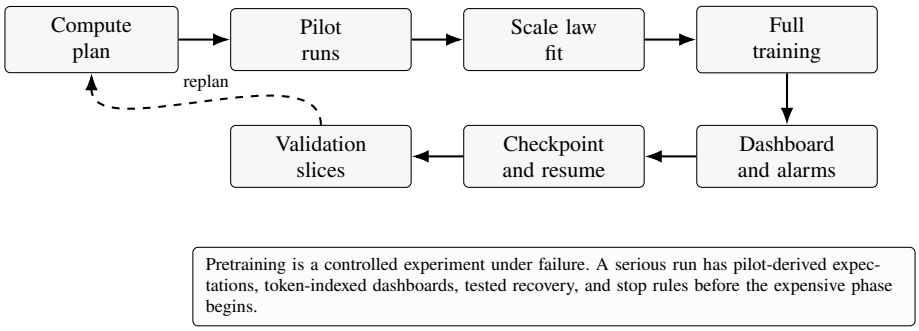

Fig. 6.1 Pretraining as an experimental control loop. Scaling-law planning, pilot runs, checkpoint recovery, and validation slices are not administrative extras; they determine whether the final loss curve is scientifically interpretable.

The recipe is a coupled choice of architecture, token budget, context length, data mixture, optimizer, precision, hardware layout, and stopping rule. Scaling-law studies made this coupling explicit by showing predictable loss trends with model size, data size, and compute over broad ranges [68]. Compute-optimal work then showed that many early large models were too parameter-heavy for the number of tokens they saw, shifting modern practice toward training smaller or similarly sized models on far more data [56]. Open model reports reinforce the point: LLaMA-class models are not only architectural artifacts; they are also data and optimization artifacts [165, 166, 48].

A practical pretraining plan begins with a units table. Let N be trainable parameters, D be training tokens, T be sequence length, B_μ be micro-batch size per device, G be the number of gradient-accumulation micro-steps, and P be the data-parallel world size. The global token batch is

$$B_{\text{tok}} = B_\mu \, T \, G \, P. \tag{6.2}$$

The number of optimizer updates is approximately D/B_{tok}. For dense decoder-only Transformers, a common planning approximation is that training costs about $6ND$ floating-point operations, excluding attention details, embedding/output costs, recomputation, data loading, and system inefficiency. The approximation is useful because it gives the right first-order economics: doubling either parameters or tokens roughly doubles training compute, while increasing context length changes both model-side attention cost and systems-side activation memory.

Figure 6.1 shows why this planning loop has to include pilot runs, dashboards, recovery, and validation from the start.

6.2 Objective Accounting and Data Order

The loss is only meaningful when tokenization, masking, and data provenance are controlled. Two models can report the same numerical loss while using different tokenizers, different document separators, different treatment of repeated text, or different valida-

tion mixtures. Perplexity is therefore most useful within a fixed experimental family. Across model families, it should be interpreted as a diagnostic rather than as a universal capability score.

Packed sequence training converts documents into fixed-length blocks. This improves accelerator utilization because every position participates in a dense matrix computation. It also creates subtle boundary choices. If document boundaries are ignored, the model may learn unrealistic transitions across unrelated documents. If every document starts with a boundary token, the model receives a distributional signal about resets, but the vocabulary and tokenizer must reserve and consistently use that marker. If samples are packed with loss masking across boundaries, the data loader must keep masks synchronized with token ids after shuffling and distributed partitioning.

Data order is an optimization variable. A uniform shuffle over all tokens is simple, but large corpora are mixtures: web pages, books, code, papers, multilingual text, math, dialogue, and synthetic data. Mixture weights determine both what the model learns and how stable the loss becomes. Code and math can produce different gradient statistics from casual web text; low-quality duplicates can lower training loss while harming downstream robustness; benchmark-like documents can inflate evaluation through contamination [153, 180]. A serious run keeps a manifest of mixture weights, token counts, deduplication policy, filtering policy, and validation splits.

The validation set should be fixed before the run and should be separated by source, not merely by random token slice. Random slices from a large web corpus may share near-duplicate documents with training. A useful validation dashboard reports loss by mixture component: general web, code, math, books, academic text, multilingual buckets, and any domain-specific data. When a global loss spike occurs, per-component losses often reveal whether the problem is optimizer instability, data corruption, a single bad shard, or a distribution shift introduced by a curriculum boundary.

6.2.1 Auxiliary Multi-Token Prediction

The default causal objective predicts the next token. Some recent large-model recipes add auxiliary prediction heads that ask the model to predict tokens farther in the future. DeepSeek-V3 describes multi-token prediction (MTP) as an auxiliary training objective: the model still learns ordinary next-token prediction, but additional modules predict x_{t+2}, x_{t+3}, or other future positions from intermediate representations [34]. A simplified loss is

$$\mathcal{L}_{\text{pretrain}} = \mathcal{L}_{\text{NTP}} + \sum_{j=1}^{J} \lambda_j \mathcal{L}_{\text{MTP},j},$$

where $\mathcal{L}_{\text{MTP},j}$ predicts a future token offset and λ_j controls how much the auxiliary task shapes the shared representation.

MTP should not be confused with a generic serving speedup. An auxiliary head may improve representations during pretraining and then be removed before deployment; a parallel-decoding head may remain in the model and change inference. The training

Table 6.1 Pretraining bookkeeping that should be fixed before a run starts.

Item	Why It Matters	Failure Mode
Tokenizer and boundary tokens	Defines the loss units and document transition signal	Perplexity cannot be compared across runs
Loss mask	Specifies which positions train the model	Padding, separators, or prompt tokens accidentally dominate gradients
Mixture weights	Controls the distribution being learned	A global loss hides regressions in code, math, or low-resource languages
Shard order and seed	Determines reproducibility and restart behavior	A resumed run silently repeats or skips data
Validation sources	Separates optimization diagnostics from benchmark claims	Near-duplicate validation makes the run look healthier than it is
Contamination checks	Protects downstream evaluation	Public benchmark text is learned during pretraining

report should therefore state whether future-token heads are used only as losses, whether they share embeddings or output heads, how their gradients are weighted, and whether they survive into the released checkpoint. Auxiliary objectives are part of objective accounting in the same sense as data mixture and masking: they change what evidence each token contributes to the run.

Table 6.1 lists the bookkeeping choices that should be fixed before the run starts.

6.3 AdamW and Parameter Groups

Most modern large language model pretraining uses AdamW or a close variant. Adam maintains exponential moving averages of first and second moments of the stochastic gradient [74]. For gradient $g_t = \nabla_\theta \mathcal{L}_t(\theta_t)$,

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \tag{6.3}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \tag{6.4}$$

with bias-corrected estimates $\hat{m}_t = m_t/(1-\beta_1^t)$ and $\hat{v}_t = v_t/(1-\beta_2^t)$. A simplified AdamW update is

$$\theta_{t+1} = \theta_t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \eta_t \lambda \theta_t, \tag{6.5}$$

where η_t is the learning rate and λ is decoupled weight decay [102]. The word "decoupled" is important. Classical L_2 regularization adds $\lambda\theta$ to the gradient, so the adaptive denominator changes the effective regularization. AdamW applies decay as a separate multiplicative shrinkage term, which makes the meaning of weight decay less entangled with gradient scale.

Parameter groups are part of the algorithm, not cosmetic code. Common practice applies weight decay to large weight matrices but excludes biases, normalization parameters, and sometimes embeddings. The reason is structural: a normalization scale or bias does not behave like a high-dimensional weight matrix whose norm should be regularized. A typical implementation builds two groups:

$$\Theta_{\text{decay}} = \{W : \dim(W) \geq 2\}, \qquad \Theta_{\text{nodecay}} = \Theta \setminus \Theta_{\text{decay}}. \tag{6.6}$$

The exact rule should be logged, because small changes can affect reproducibility across codebases.

The Adam hyperparameters also encode assumptions about gradient noise. Many large language model recipes use β_1 near 0.9 and β_2 lower than the original Adam default of 0.999, often around 0.95 or 0.98, so the second-moment estimate adapts faster during long runs with changing data mixtures. The ϵ value is not merely a numerical afterthought in low precision: it affects the denominator when second moments are small. Any comparison of optimizer variants should state $(\beta_1, \beta_2, \epsilon)$, weight decay, clipping rule, schedule, batch size, and precision.

6.4 Schedules, Warmup, and Batch Size

The learning-rate schedule controls how aggressively the run spends optimization budget. A common schedule warms up linearly, decays with a cosine curve, and ends at a nonzero minimum:

$$\begin{aligned} \eta_t &= \eta_{\max} \frac{t}{t_w}, && \text{for } t < t_w, \\ \eta_t &= \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\left(1 + \cos\left(\pi \frac{t - t_w}{t_d - t_w}\right)\right), && \text{for } t_w \leq t \leq t_d, \\ \eta_t &= \eta_{\min}, && \text{for } t > t_d. \end{aligned} \tag{6.7}$$

Warmup prevents the first few thousand steps from taking full-size adaptive updates before moment estimates and activation scales have settled. Cosine decay gradually reduces update size without a sharp discontinuity. Some production runs reserve a final annealing phase on a changed mixture, but such phases should be treated as new experiments: changing the data distribution near the end can improve targeted evaluations while obscuring what the base model learned from the main corpus.

Batch size is best measured in tokens, not examples. A batch of eight sequences at context length 8192 contains as many token positions as sixty-four sequences at context length 1024. Larger token batches reduce gradient noise and improve device utilization, but they also reduce the number of optimizer updates for a fixed token budget. Gradient accumulation simulates a larger batch when the micro-batch that fits in memory is small. With data parallelism, the effective batch is the product of micro-batch, sequence length, accumulation steps, and replicas. If that product changes, the learning-rate schedule and warmup should be reconsidered rather than inherited blindly.

A useful diagnostic is the tokens-per-update curve together with loss. If a run changes sequence length, packing efficiency, data-parallel size, or accumulation steps after resumption, the apparent update count no longer corresponds to the same number of tokens. For large runs, logs should use token count as the primary training-time axis and optimizer step as a secondary axis.

6.5 Gradient Clipping and Stability

Large pretraining runs fail in recognizable ways: loss spikes, NaNs, diverging gradient norms, stuck data loaders, corrupted checkpoints, and sudden throughput drops. Gradient clipping is a simple guardrail. Global norm clipping rescales gradients when

$$\|g\|_2 = \sqrt{\sum_i \|g_i\|_2^2} \tag{6.8}$$

exceeds a threshold c, replacing g with $g \cdot c/(\|g\|_2 + \epsilon)$. Clipping does not fix a bad schedule or bad data, but it can keep a rare large batch from destroying the optimizer state. The clipped fraction should be logged; constant clipping means the threshold or learning rate is probably wrong, while rare clipping during outlier batches may be acceptable.

Stability also depends on architecture. Pre-normalization improves gradient flow in deep decoders. RMSNorm, residual scaling, careful initialization, and gated feed-forward layers change activation distributions. Long context increases attention score ranges and KV-cache sizes. Mixture-of-experts models add router logits and load-balancing losses, so a stable dense recipe may not transfer directly to a sparse model [34]. Optimizer settings should therefore be tied to the architecture version and not recorded as generic "LLM defaults."

Mixed precision is a second source of instability. FP16 has limited exponent range, so loss scaling became a standard technique for avoiding gradient underflow [108]. BF16 has fewer mantissa bits but a wider exponent range, which makes it attractive for large Transformer training because it often avoids explicit loss scaling. FP8 and other lower-precision formats can improve throughput but require careful scaling, accumulation, and validation against higher-precision baselines. A publishable training report states which tensors are stored, multiplied, reduced, and accumulated in which precision.

FP8 is not one undifferentiated format. The deep-learning FP8 proposal separates E4M3 and E5M2 encodings, commonly using the wider mantissa of E4M3 for weights and activations and the wider exponent range of E5M2 for gradients [109]. Operations still accumulate or produce higher-precision values, and tensors usually need scaling before conversion into FP8. Therefore an FP8 claim should report tensor roles, scaling granularity and update rule, saturation or rounding behavior, accumulation dtype, and the higher-precision baseline it was checked against.

Logit scale is another stability surface. Baichuan 2 reports two practical mechanisms: normalizing the output head so logits depend less on unstable embedding norms, and

adding a max-z loss that penalizes very large maximum logits [184]. A simplified form is

$$z_{\max} = \max_i \ell_i, \qquad \mathcal{L}_{\text{max-}z} = \lambda z_{\max} z_{\max},$$

where ℓ_i are the vocabulary logits. This kind of auxiliary penalty is not about lowering training loss directly. It makes downstream decoding less brittle to repetition penalties, logit processors, and softmax saturation. If such a term is used, the report should state its coefficient, whether it is applied throughout training or only during a phase, and how generation behavior changes when it is removed.

Loss spikes need triage rather than superstition. The first question is whether the spike is local to one worker, one data shard, one mixture component, or the whole job. The second is whether weights, optimizer moments, or both have been corrupted. If only a bad batch caused a transient spike, skipping or clipping may be enough. If optimizer state contains infinities, resuming from a checkpoint before the spike is safer. If the validation loss changes permanently, the run has entered a different region of parameter space and should be treated as a new branch.

6.6 Checkpointing and Reproducibility

A checkpoint is more than model weights. For reproducible pretraining it must contain model parameters, optimizer state, scheduler state, random-number generator states, data-loader position, tokenizer identity, configuration, and enough sharding metadata to reconstruct the model. AdamW stores two moment tensors per parameter, so optimizer state can be larger than the weights themselves. Distributed systems may save sharded checkpoints to avoid gathering a full model on one host; the checkpoint format then becomes part of the system contract.

Checkpoint cadence is a cost tradeoff. Frequent checkpoints reduce lost work after failure but consume I/O bandwidth and storage. In a large cluster, writing checkpoints can stall training if all workers flush simultaneously to a shared filesystem. A practical plan staggers writes when possible, keeps a small rolling window of recent recovery checkpoints, and separately exports less frequent archival checkpoints for evaluation. The archival checkpoint should include enough metadata to identify exactly which data and code produced it.

Intermediate checkpoints can also be scientific artifacts. Baichuan 2's release plan included token-indexed 7B checkpoints from roughly 220 billion tokens through the final 2.64 trillion-token model, making it possible to inspect how benchmark slices improve or plateau during training [184]. A checkpoint series is most useful when every checkpoint is tied to the same tokenizer, data manifest, evaluation harness, and token count. Otherwise training dynamics can be mistaken for changes in evaluation protocol.

Resumption should be tested before the expensive run. A small dry run should train for a few updates, save, resume, and verify that loss and learning rate continue on the expected trajectory. This catches common errors: forgetting optimizer state, resetting the scheduler, repeating data shards, changing dropout seeds, or loading weights under a

slightly different architecture. The dry run is not administrative overhead; it is a systems test of the scientific claim.

6.7 Compute-Optimal Planning

Scaling laws are planning tools, not laws of nature. They are empirical fits over specific architectures, tokenizers, corpora, and training procedures. The useful lesson is not that a single formula determines the next model, but that compute should be allocated deliberately between parameters and tokens. If a model is too large for the available token budget, it may have impressive capacity but underdeveloped weights. If a model is too small for the token budget, extra data may produce diminishing returns because capacity becomes the bottleneck [56].

A planning worksheet usually includes:

1. the target model family and parameter count N;
2. the token budget D by mixture component;
3. the sequence length and expected packing efficiency;
4. the estimated FLOPs, $6ND$ as a baseline plus attention and overhead adjustments;
5. expected model FLOPs utilization (MFU) or hardware utilization;
6. wall-clock time under realistic failure and checkpoint overhead;
7. validation checkpoints and go/no-go criteria.

The worksheet should be updated with measured throughput from a scaled-down run. Small-batch microbenchmarks often overestimate full-run throughput because they omit data loading, checkpointing, network contention, stragglers, validation, and restarts.

Undertraining is a common interpretation trap. A larger model trained on too few tokens can beat a smaller model on some few-shot benchmarks while having worse cost-performance. Conversely, a smaller model trained longer can be easier to serve and adapt. Model reports such as LLaMA, Llama 2, and Llama 3 should be read with this tradeoff in mind: parameter count alone is a poor proxy for training investment or deployment cost [165, 166, 48].

Table 6.2 collects the first-order quantities that make such planning auditable.

6.8 Monitoring, Evaluation, and Stop Rules

Training dashboards should separate optimization health from model quality. Optimization health includes training loss, validation loss by mixture, gradient norm, update norm, learning rate, clipping rate, tokens processed, throughput, memory reserved, data-loader wait time, all-reduce time, checkpoint time, and error counts. Model quality includes held-out tasks, generation samples, code tests, math tests, multilingual probes, safety probes, and contamination-aware benchmark suites. The first set answers “is the run functioning?” The second answers “is the artifact useful?”

Table 6.2 First-order compute and memory quantities used in pretraining planning. Constants vary by architecture and implementation, so these formulas are planning approximations.

Quantity	Approximation	Interpretation
Training FLOPs	$6ND$	Dense decoder baseline for forward and backward passes
Token batch	$B_\mu TGP$	Tokens consumed per optimizer update
Optimizer steps	$D/(B_\mu TGP)$	Schedule length should track tokens as well as steps
Parameter memory	$N\times$ bytes per parameter	Weight storage before gradients and optimizer states
Adam state memory	$2N\times$ state bytes	First and second moments dominate training memory
Activation memory	Proportional to layers, B_μ, T, hidden size	Often the reason for checkpointing or smaller micro-batches

Loss prediction can detect problems early. If prior scaling fits or smaller pilot runs predict a smooth loss curve, deviations deserve explanation. A loss above prediction may indicate poor data quality, an overly aggressive learning rate, insufficient warmup, a bug in masking, or lower-than-expected batch quality. A loss below prediction may be good news, but it can also signal validation leakage, duplicate-heavy data, or a tokenizer difference. The direction of surprise is less important than whether the team can explain it.

Stop rules should be written before the run. Examples include a maximum token budget, a target validation loss, a budget cap, or a decision checkpoint where marginal loss improvement no longer justifies cost. Without a stop rule, teams are tempted to continue because the curve still decreases, even when the deployment target would be better served by spending compute on data cleaning, post-training, evaluation, or inference optimization.

6.9 A Quantitative Run Card

A pretraining run should leave behind a run card that can be audited without private dashboards. At minimum, record

$$C_{\text{run}} = (N, D, T, B_{\text{tok}}, \eta_{\text{mfu}}, \eta_{\text{pack}}, \eta_{\text{fail}}, \mathcal{V}, \mathcal{S}), \tag{6.9}$$

where η_{mfu} is achieved model FLOPs utilization, η_{pack} is packing efficiency, η_{fail} is the fraction of wall-clock time lost to failures and checkpointing, $\mathcal{V}$ is the validation-slice set, and $\mathcal{S}$ is the stop-rule set. A model report that gives N and benchmark scores but omits D, T, tokenizer, data mixture, training efficiency, or validation-slice behavior is not sufficient evidence for a reproducible training claim.

6.10 Implementation Notes

The local implementation exercises that motivate this chapter use a progression from a small Transformer training loop to GPT-style and LLaMA-style pretraining scripts. The textbook lesson is not the literal code but the invariants those scripts expose: zero gradients before backward, scale loss during accumulation, clip after unscaling and before the optimizer step, step the scheduler consistently, save optimizer state, and log tokens per second. In a distributed run, only some workers should perform human-readable logging, but all workers must agree on seeds, data partitions, and checkpoint barriers.

Several bugs are easy to miss in review. If the target tensor is shifted incorrectly, the model may learn to copy the current token rather than predict the next. If padding is not masked, the model can achieve low loss by learning padding statistics. If loss is divided by accumulation steps after the backward pass instead of before, gradient scale changes with accumulation. If validation uses training dropout or an unfrozen data order, the loss becomes noisy for reasons unrelated to generalization. If the final partial accumulation step is mishandled, the last updates of an epoch can have a different effective learning rate.

6.11 Key Terms

AdamW Adam with weight decay applied as a separate parameter shrinkage term rather than as an adaptive gradient penalty.

Gradient accumulation Multiple forward/backward micro-steps whose gradients are summed before one optimizer step.

Global norm clipping Rescaling all gradients together when their combined norm exceeds a threshold.

FP8 scaling The tensor scaling policy needed to map higher-precision values into FP8 ranges before low-precision matrix operations.

MFU Model FLOPs utilization, the ratio of achieved model-relevant FLOPs to hardware theoretical peak.

Multi-token prediction An auxiliary objective that predicts future tokens beyond the immediate next token, often to shape representations or support later decoding research.

Packed sequence A fixed-length token block formed by concatenating or slicing documents for efficient training.

Warmup An initial schedule phase that increases learning rate gradually while optimizer moments and activations settle.

6.12 Exercises

1. A model has $N = 7$ billion parameters and is trained on $D = 1.4$ trillion tokens. Estimate dense training FLOPs with the $6ND$ rule. If the cluster sustains 1.2×10^{18} FLOPs/s on model work, estimate ideal wall-clock days before checkpointing and failures.
2. Suppose $B_\mu = 4$, $T = 4096$, $G = 16$, and $P = 128$. Compute tokens per optimizer update and the number of optimizer updates for 300 billion tokens. Explain how the answer changes if the context length doubles while the token budget stays fixed.
3. Write pseudocode for one accumulation cycle using mixed precision, loss scaling if needed, global norm clipping, AdamW, and a cosine schedule. State exactly where gradients are divided by the accumulation factor.
4. Write an FP8 run-card row for one training experiment. State which tensors use E4M3 or E5M2, where accumulation stays higher precision, how scaling factors are chosen, and which BF16 or FP16 baseline validates the result.
5. Design a validation dashboard with at least six mixture-specific losses. For each loss, name one data or optimization failure it could reveal before global validation loss does.
6. Add an auxiliary two-token-ahead loss to the causal objective. State which hidden state predicts the future token, how you weight the auxiliary loss, and what you would remove before inference if the head is training-only.
7. A run resumes from checkpoint and immediately reports a lower training loss but a higher validation loss. List four possible causes and the evidence you would inspect for each.
8. Compare two plans with the same compute budget: a larger model on fewer tokens and a smaller model on more tokens. Describe one deployment scenario where each plan could be the better choice.

Chapter 7
Distributed Training Systems

Abstract This chapter explains the systems layer that makes large-scale training feasible. It develops memory accounting for parameters, gradients, optimizer states, activations, and communication buffers; then covers data parallelism, fully sharded training, ZeRO-style optimizer partitioning, tensor parallelism, pipeline parallelism, expert parallelism, precision, collectives, throughput accounting, checkpointing, and operational reliability.

Chapter contract.

The reader should leave this chapter able to translate a model and batch plan into memory, communication, and throughput constraints, choose a parallelism strategy for those constraints, and recognize when an apparent modeling problem is actually a systems or reliability problem.

7.1 Why Training Systems Are Part of the Model

Large-scale pretraining is not obtained by placing a textbook optimizer inside a larger loop. The distributed system changes what batch size is affordable, how sequence length is chosen, which precision is safe, how often checkpoints can be written, and how quickly failures can be recovered. A model recipe that is mathematically clear but cannot fit in memory or communicate efficiently is not a runnable recipe.

The central design problem is to map a computation graph onto accelerators with limited memory and finite interconnect bandwidth. If the model fits on one GPU but the desired batch does not, data parallelism may be enough. If optimizer state is too large, sharding is needed. If a single layer's matrix multiplications do not fit or do not run efficiently, tensor parallelism becomes relevant. If depth or activation memory dominates, pipeline parallelism or activation checkpointing may be the right lever. If the model is sparse, expert parallelism adds a routing and load-balancing problem. Modern

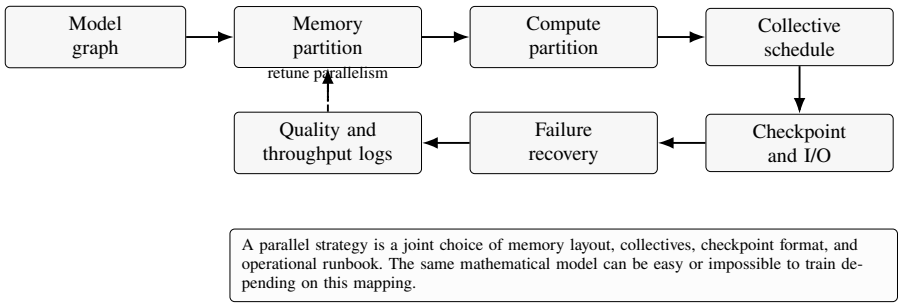

Fig. 7.1 Distributed training as a systems mapping problem. Data, FSDP/ZeRO, tensor, pipeline, and expert parallelism move memory and communication pressure to different parts of the runtime.

frontier training commonly composes several of these strategies rather than choosing only one [115, 34].

The discipline of this chapter is accounting. Each strategy moves bytes, computation, or idle time from one place to another. It is not enough to say that a method "saves memory." A useful systems design says which bytes are saved, which collectives are added, how overlap is achieved, what happens during checkpointing, and which failure mode becomes more likely.

Figure 7.1 frames those choices as one systems mapping problem rather than independent tricks.

7.2 Memory Accounting

For dense training with AdamW, the persistent state per parameter usually includes the parameter value, a gradient, and two optimizer moments. If parameters and gradients are stored in BF16 and Adam moments are stored in FP32, a rough per-parameter budget is

$$2 \text{ bytes} + 2 \text{ bytes} + 4 \text{ bytes} + 4 \text{ bytes} = 12 \text{ bytes}. \tag{7.1}$$

Some implementations also keep FP32 master weights, which raises the total. A 70-billion-parameter dense model can therefore require hundreds of gigabytes before activations, temporary buffers, CUDA allocator fragmentation, or communication workspaces are counted. This is why single-device reasoning about parameter count is misleading.

Activation memory is often the limiting term for long context or large micro-batches. During backpropagation, the system needs enough intermediate values to compute gradients. For a decoder with L layers, hidden size H, micro-batch B_μ, and sequence length T, residual activations alone scale like $O(LB_\mu TH)$, while attention internals can add terms that depend on heads and sequence length. Memory-efficient attention kernels reduce some temporary storage, but they do not remove the need to retain or recompute the values required for backward.

Table 7.1 Major memory categories during dense large language model training.

Category	Scaling Driver	Typical Mitigation
Parameters	N and parameter precision	Tensor parallelism, FSDP, ZeRO-3, CPU or NVMe offload
Gradients	N and gradient precision	ZeRO-2/3, reduce-scatter, gradient accumulation
Optimizer states	$2N$ or more for Adam-style methods	ZeRO optimizer partitioning, lower-precision states, optimizer choice
Activations	$LB_\mu TH$ plus attention intermediates	Smaller micro-batch, activation checkpointing, sequence parallelism
Communication buffers	Collective size and overlap strategy	Bucket sizing, topology-aware process groups, overlap tuning
Temporary workspaces	Kernel and allocator behavior	Memory headroom, fused kernels, static shapes where possible

Temporary memory is the least visible category and one of the most common sources of out-of-memory errors. Matrix multiplication libraries allocate workspaces; attention kernels may allocate scratch buffers; distributed collectives may require staging buffers; model code may create transient copies through layout conversions; and memory fragmentation can prevent a large allocation even when total free memory looks sufficient. Production training systems leave headroom rather than filling the device to the last megabyte.

Table 7.1 separates the major memory categories so later parallelism choices can be tied to concrete bytes.

7.3 Data Parallelism

Data parallelism replicates the model on each worker, gives each replica a different micro-batch, and averages gradients before the optimizer step. In synchronous data parallel training, the mathematical update is close to a larger batch on one device, assuming identical numerics and deterministic reduction. The main collective is an all-reduce over gradients:

$$g = \frac{1}{P} \sum_{r=1}^{P} g^{(r)}, \tag{7.2}$$

where P is the data-parallel world size. Distributed data parallel implementations usually bucket gradients so communication for early layers can overlap with backward computation for later layers.

The strength of data parallelism is simplicity. It preserves the local model code and increases token throughput as long as communication is not dominant. The weakness is memory replication: every worker stores all parameters, gradients, and optimizer states. For small and medium models this is acceptable. For very large models, data parallelism alone runs out of memory before it runs out of compute.

Large data-parallel worlds also change optimization. If the global token batch grows with the number of workers, the number of optimizer updates for a fixed token budget falls. Learning-rate schedules, warmup, and gradient noise properties must be retuned. Scaling out a run is therefore both a systems change and an optimization change.

7.4 ZeRO and Fully Sharded Data Parallel

ZeRO-style training partitions redundant data-parallel state across workers [141]. The usual stages are easiest to remember by what they shard:

Stage 1 Shard optimizer states.
Stage 2 Shard optimizer states and gradients.
Stage 3 Shard optimizer states, gradients, and parameters.

The memory savings can be dramatic because Adam moments and gradients no longer exist in full on every data-parallel worker. The cost is communication: workers must gather parameter shards before computation, reduce or reduce-scatter gradients after backward, and coordinate optimizer updates.

Fully Sharded Data Parallel (FSDP) is a widely used PyTorch implementation of this idea, especially close to ZeRO-3 in spirit [201]. A module wrapped with FSDP stores only local parameter shards most of the time. Before the module's forward or backward computation, workers all-gather the full parameters needed for that module; after use, full parameters can be freed. Gradients are reduce-scattered so each rank keeps only its shard. Nested wrapping controls the granularity of gathering and freeing.

PyTorch's newer FSDP2 interface makes that contract more explicit through `fully_shard` and `DTensor` parameters [133]. During initialization, parameters are converted in place from ordinary tensors to sharded `DTensor`s on a device mesh. Forward and backward hooks all-gather parameters before computation, expose ordinary tensors while the module runs, and convert them back to sharded tensors after the computation. This means the optimizer must be constructed over the `DTensor` parameters, ordinary model calls with inputs must trigger the hooks, and a bottom-up wrapping order controls all-gather groups. FSDP2 also changes the checkpoint expectation: instead of treating a full state dict as the default export path, the run should record how sharded `DTensor` state is resharded, materialized, or handed to PyTorch Distributed Checkpoint APIs [131].

This is memory sharding, not the same thing as tensor-parallel compute sharding. A ZeRO-3 or FSDP worker may store only a shard between module calls, but the module still needs the full parameter view when its local computation runs. The method reduces persistent model-state memory; it does not by itself split a matrix multiplication into smaller matrix multiplications across ranks. Confusing these two facts leads to bad designs: ZeRO/FSDP is the right first lever when optimizer state or weights do not fit, while tensor parallelism is the right lever when individual layer computation or layer-local memory is the bottleneck.

The later ZeRO family shows what happens after ordinary sharding moves the bottleneck from memory capacity to communication and data movement. ZeRO-Offload

moves selected optimizer state, gradients, and computation to CPU memory to fit larger models on limited GPU memory, but it only helps when transfer and CPU work overlap with accelerator compute [144]. ZeRO-Infinity extends the same idea to heterogeneous GPU, CPU, and NVMe memory, making storage hierarchy and prefetch granularity part of the training algorithm [142]. ZeRO++ instead targets communication volume with block-quantized weight all-gather, hierarchical partitioning, and quantized gradient exchange [170]. The practical lesson is that "fits in memory" is not the end of the design: report PCIe or NVLink bandwidth, host-memory pressure, NVMe throughput if used, collective volume, overlap efficiency, and whether accuracy was checked against a non-compressed baseline.

Baichuan 2 illustrates a pragmatic hybrid layout for dense multilingual models: tensor parallelism within an eight-GPU machine, ZeRO-style data parallelism across machines, tensor splitting for memory-heavy operations such as large-vocabulary cross entropy, topology-aware rank placement, and hybrid or hierarchical parameter partitioning to reduce interconnect pressure at larger scale [184]. The transferable lesson is the placement rule, not the exact cluster. Keep layer-local tensor-parallel communication inside a fast island when possible, shard redundant state across the wider data-parallel group, and treat rank mapping and checkpoint export as part of the training recipe.

The important design question is granularity. If wrapped units are too large, peak memory rises because full parameters for a large region are materialized at once. If units are too small, collective overhead and scheduling overhead increase. Transformer block wrapping is a common compromise. Auto-wrapping policies should be inspected, not trusted blindly, because shared embeddings, tied output heads, adapters, and mixture-of-experts layers can violate simple assumptions.

FSDP also affects checkpointing. A full state dict is convenient for evaluation and sharing but expensive to materialize. A sharded state dict is efficient for resuming training but tied to metadata about world size, parameter flattening, and sharding layout. A robust training pipeline tests both resumption and export. It should be possible to recover the job after a node failure and separately export a canonical checkpoint for inference or post-training.

ZeRO-3 plus parameter-efficient tuning adds another export contract. The training checkpoint may contain partitioned optimizer and adapter state rather than a directly loadable Hugging Face model. A release pipeline therefore has to reconstruct a full-precision state dict from the ZeRO shards, extract the PEFT adapter keys, preserve any saved modules such as a reward-model score head, and then decide whether to publish an adapter or merge it into the base model. The tokenizer and chat template must be copied with the exported artifact, because an adapter that loads successfully against the wrong base tokenizer can still produce invalid generations.

7.5 Tensor Parallelism

Tensor parallelism splits individual layer computations across devices. Megatron-LM popularized practical one-dimensional tensor parallel layouts for Transformer language models [154]. For a linear layer $Y = XW$, column parallelism splits W by output

columns, so each device computes a shard of Y. Row parallelism splits W by input rows, so devices compute partial outputs that must be summed. MLPs and attention projections can be arranged so some collectives are delayed or paired with the next operation.

The communication placement is the important part. In a two-layer MLP, the first projection is often column-parallel: each rank produces a different shard of the expanded hidden dimension, and the elementwise activation can run locally. The second projection is then row-parallel: each rank consumes its local hidden shard, computes a partial output in the original hidden dimension, and an all-reduce sums the partial outputs. Attention follows a similar idea by partitioning query, key, and value projections by attention head, then using a row-parallel output projection whose partial outputs must be reduced.

The advantage is that very large layers can run when a single device cannot store or multiply the full matrices efficiently. Tensor parallelism also increases arithmetic capacity for one model replica. The cost is frequent communication inside the forward and backward pass. These collectives are latency-sensitive and topology-sensitive, so tensor-parallel groups are usually kept within a high-bandwidth island such as GPUs connected by NVLink or within one server. Spanning tensor parallelism across slower inter-node links can destroy utilization.

Some libraries expose higher-dimensional variants beyond Megatron-style one-dimensional row and column partitioning. The OSLO tensor-model-parallel tutorial, for example, describes one-dimensional, two-dimensional, 2.5-dimensional, and three-dimensional tensor partitioning modes and treats tensor-parallel size, mode, depth, and process-group layout as explicit runtime configuration [185]. These modes are not interchangeable switches: they change collective patterns, checkpoint layout, and the set of valid process groups. The same tutorial also exposes several operational constraints: the model is created on CPU before sharding, `tensor_parallel_size` must be positive, no larger than the available GPU count, and a power of two, `tensor_parallel_depth` is only meaningful for the 2.5D mode, and the shown tensor-parallel path does not mix with pipeline parallelism. Its default `save_pretrained` output is a rank-sharded checkpoint with tensor, pipeline, and expert rank identifiers; exporting an ordinary checkpoint requires an explicit merge step. A training run should therefore log the tensor-parallel degree, mode, depth if used, data/pipeline/expert parallel degrees, rank mapping, and whether the saved checkpoint is sharded by tensor-parallel rank or exported as a merged model.

PyTorch's newer tensor-parallel tutorials express the same idea through `DeviceMesh`, `DTensor`, and module-level parallel styles [91, 196]. A `DeviceMesh` names the process-group dimensions, so a two-dimensional layout can put tensor parallelism on an intra-host mesh and FSDP on an inter-host mesh instead of requiring hand-built process groups. A parallelization plan then maps submodules to `ColwiseParallel`, `RowwiseParallel`, `SequenceParallel`, or explicit input and output layout transforms. This makes shape contracts visible: a column-sharded Q/K/V projection shards the head dimension, so reshape operations, RoPE buffers, and local head counts must either be DTensor-aware or be rewritten for local shapes. If the model cannot even be instantiated in CPU memory before sharding, meta-device or layer-by-layer initialization becomes part of the recipe, not just an implementation detail.

Attention introduces additional choices. Query, key, and value projections can be partitioned by head because different heads are mostly independent until output projection. Sequence parallelism splits some activation tensors along the sequence dimension to reduce replicated activation memory, but it adds collectives around operations that need full sequence or hidden dimensions. The implementation must align partitioning with kernel expectations; otherwise layout conversions can erase the intended savings.

Sequence, loss, and context parallelism are worth separating. Sequence parallelism usually keeps block inputs and outputs sharded on the sequence dimension, applies it to normalization or dropout layers, and inserts layout conversions before attention and feed-forward modules. Loss parallelism keeps the large vocabulary logits sharded and computes cross entropy without first all-gathering full logits onto every rank, which is important for large vocabularies and tied output heads. Context parallelism instead targets very long sequences by sharding Q/K/V or other sequence buffers and replacing ordinary scaled-dot-product attention with a ring-attention variant; the PyTorch prototype exposes both all-gather and all-to-all KV rotation paths [177]. Position-dependent buffers such as RoPE frequencies must be included in the same sequence sharding plan, or the attention output can be numerically wrong even though tensor shapes still match.

7.6 Pipeline Parallelism

Pipeline parallelism partitions layers into stages. Stage 0 processes early layers, sends activations to stage 1, and so on. Backward sends gradients in the reverse direction. GPipe introduced a clean micro-batch pipeline approach for training very deep networks [62]. In language-model training, pipeline parallelism is often combined with tensor and data parallelism for large models [115].

Recent PyTorch pipeline support exposes this as a split frontend plus a distributed runtime rather than only as hand-written send/receive code [132]. The package is still documented as alpha, but its shape is useful for run-card design: the frontend records model partitions and data-flow relationships, while the runtime handles micro-batch splitting, scheduling, communication, and gradient propagation. Schedules such as GPipe, 1F1B, interleaved 1F1B, and Looped BFS are not only performance switches; they define activation lifetime, optimizer-step timing, cross-host traffic, and checkpoint reconstruction. TorchTitan's Llama training stack shows the same composition pressure in practice by combining FSDP2, tensor parallelism, pipeline parallelism, context parallelism, activation checkpointing, distributed checkpointing, float8 support, structured per-rank logging, and checkpointable data loading [164].

The basic problem is the pipeline bubble. If there are S stages and only one micro-batch, most stages are idle most of the time. Splitting a batch into M micro-batches fills the pipeline:

$$\text{bubble fraction} \approx \frac{S-1}{M+S-1} \tag{7.3}$$

for a simple schedule. More micro-batches reduce idle time but increase activation bookkeeping and may change effective batch-size constraints. Interleaved schedules

assign multiple stage chunks to a device to improve load balance, but they increase scheduling complexity.

Stage balance is critical. A pipeline runs at the speed of its slowest stage. Embedding layers, output projections, attention-heavy long-context layers, and MoE layers can make equal layer counts unequal in runtime. A good partitioner measures real execution time and memory rather than assuming identical blocks. It also accounts for activation transfer sizes between stages.

Modern sparse models add another reason to treat the schedule as a first-class algorithm. DeepSeek-V3 reports a DualPipe-style pipeline schedule designed to overlap forward, backward, and communication work in a setting that also uses expert and pipeline parallelism [34]. The general lesson is not that every system should copy one named schedule. It is that pipeline efficiency depends on micro-batch count, stage balance, activation transfer, all-to-all traffic, and whether communication can be overlapped with useful computation.

Pipeline parallelism changes failure recovery and debugging. A NaN generated in one stage may only be observed downstream. A shape bug may depend on a boundary between stages. Logging must include stage rank, micro-batch id, and schedule step. Checkpoints must reconstruct partitioned modules correctly, especially if the model is later exported without pipeline partitioning.

7.7 Expert Parallelism

Mixture-of-experts models replace some dense feed-forward computation with a set of experts and a router. Each token is sent to one or more experts, so total parameters can be much larger than active parameters per token. Expert parallelism places different experts on different devices and uses all-to-all communication to move token representations to the devices that own the selected experts. GShard and later MoE systems made this style of conditional computation central to large sparse models [82, 34].

The systems challenge is load balance. If the router sends too many tokens to one expert, that expert becomes a straggler and may overflow its capacity. Load-balancing losses, capacity factors, token dropping policies, and auxiliary routing constraints are therefore part of the training system. They affect both optimization and throughput. A sparse model can have excellent nominal FLOPs per token and still run poorly if routing creates uneven all-to-all traffic.

Recent DeepSeek-style training material makes this concrete: routed experts, shared experts, sequence-level balancing, and expert-parallel dispatch are not separable details. A load-balance mechanism that improves one batch shape may behave differently at batch size one, during long prefill, or under skewed domain mixtures. The training dashboard should therefore show expert token counts, routing entropy, capacity overflow or rerouting, all-to-all time, and per-expert gradient activity rather than only aggregate loss.

Expert parallelism composes awkwardly with other strategies. Tensor-parallel experts require communication inside expert computation; data-parallel replicas need consistent router gradients; pipeline stages containing MoE layers may have variable

runtime; sharded checkpointing must preserve expert placement and optimizer state. This is a case where the architecture and system are inseparable.

7.8 Activation Checkpointing and Recomputation

Activation checkpointing trades compute for memory. Instead of storing every intermediate activation needed for backward, the system stores selected checkpoints and recomputes missing intermediates during the backward pass. The general technique predates modern large language models and can reduce activation memory substantially at the cost of extra forward computation [21]. In large Transformers, selective recomputation is often applied at the block level or to memory-heavy subcomponents.

The tradeoff is not simply "30 percent more compute for less memory." Recomputation interacts with pipeline schedules, tensor parallel collectives, dropout determinism, attention kernels, and compiler graph capture. If a recomputed segment includes random operations, random-number states must be replayed correctly. If the segment includes communication, recomputation may add network traffic as well as arithmetic. If the bottleneck is memory bandwidth rather than compute, recomputation can have a different cost than expected.

The practical question is which activations are worth storing. Small cheap activations can be recomputed. Expensive or communication-heavy intermediates may be worth keeping. A measured memory profile is better than a uniform policy. Modern training stacks often expose several checkpointing modes; their names are less important than the actual saved tensors and recomputed graph.

7.9 Precision and Communication

Distributed training depends on both numerical precision and communication precision. FP32 remains common for optimizer moments and some reductions. BF16 is widely used for activations and weights because its exponent range is forgiving. FP16 can be efficient but often requires loss scaling. FP8 and related formats can improve throughput and reduce communication volume, but only when scaling, accumulation, and kernel support are mature enough for the architecture and hardware [108, 34].

An FP8 claim should specify more than the format name. The report should say which tensors are stored in FP8, which matrix multiplications use FP8 inputs, which accumulations use higher precision, how scales are computed and updated, and which validation slices were compared against BF16 or FP16 baselines. For sparse models, precision interacts with router logits, load-balancing losses, expert gradients, and communication compression, so a stable dense recipe may not transfer unchanged.

Collectives are the language of distributed training:

All-reduce Every rank contributes a tensor and receives the reduced result. Used in classical data-parallel gradient averaging.

Reduce-scatter The reduction result is partitioned across ranks. Used in sharded gradients.

All-gather Shards are gathered so every rank receives the full tensor. Used before FSDP module computation.

All-to-all Each rank sends different slices to different ranks. Used heavily in expert parallelism.

Broadcast One rank sends the same tensor to others. Used for initialization and some checkpoint flows.

Point-to-point One rank sends to another rank. Used to build custom schedules, pipeline transfers, or debugging harnesses.

Scatter/Gather One rank splits a tensor across ranks, or collects rank-local tensors back to one rank. Useful for data movement and export paths, but rarely the main high-throughput training primitive.

The same collective can have different cost depending on message size, topology, contention, and overlap. For example, a small all-reduce may be latency-dominated, while a large reduce-scatter may be bandwidth-dominated.

Collective names describe semantics, not the algorithm used on the wire. An all-reduce may be implemented with a ring, tree, hierarchical intra-node plus inter-node schedule, or hardware-specific path such as NVLink/NVSwitch. A ring all-reduce is bandwidth efficient for large messages because each rank sends and receives chunks around the ring, but it is not automatically best for small latency-sensitive buckets. Training reports should therefore state the communication backend, topology, bucket sizes, and whether collectives are measured on the target cluster rather than assumed from a toy benchmark.

Overlap is the main performance technique. During backward, gradients for later layers become available before earlier layers finish. A system can launch communication for ready gradient buckets while computation continues. FSDP can prefetch upcoming parameter shards. Pipeline parallelism can overlap different micro-batches. Overlap is fragile: a bucket that is too large starts late; a bucket that is too small has high launch overhead; a dependency inserted by logging, synchronization, or shape conversion can serialize the step. Nonblocking send, receive, or collective APIs create only the opportunity for overlap; the program must delay the corresponding wait until useful independent computation has run. Ranks in the same process group must still call compatible collectives in compatible order, or a fast-looking asynchronous schedule becomes a deadlock.

7.10 Throughput Accounting

Training performance should be reported in tokens per second, time per optimizer step, and model FLOPs utilization. Tokens per second is directly tied to the training budget. Step time is useful for debugging schedules and checkpoint cadence. MFU estimates how much of the hardware's theoretical compute is used for model-relevant operations:

Table 7.2 Parallelism strategies and their primary tradeoffs.

Strategy	Main Saving	Main Cost	Best First Use
Data parallel	More token throughput	Replicated model state, gradient all-reduce	Model fits per device, batch should scale
ZeRO/FSDP	Shards optimizer, gradients, parameters	All-gather and reduce-scatter, checkpoint complexity	Model state exceeds memory
Tensor parallel	Splits large layer compute/state	Frequent intra-layer collectives	Layers too large or too slow on one GPU
Pipeline parallel	Splits depth and activation footprint	Pipeline bubbles, stage balance	Deep model cannot fit as one partition
Expert parallel	Splits sparse experts	All-to-all, load balance	MoE model with many experts
Activation checkpointing	Reduces saved activations	Recomputation and determinism constraints	Activation memory dominates

$$\text{MFU} = \text{estimated model FLOPs per step}/(\text{step time} \times \text{peak hardware FLOPs}). \quad (7.4)$$

The numerator is usually approximate, so MFU should be treated as an engineering diagnostic rather than a precise physical measurement.

End-to-end throughput includes data loading, host-to-device transfer, forward, backward, optimizer step, communication, validation, checkpointing, and restarts. A microbenchmark that measures only matrix multiplications is useful for kernel development but not for planning a training run. Conversely, a low end-to-end number does not identify the bottleneck. Profiling should separate data time, compute time, collective time, optimizer time, and I/O time.

Stragglers matter because synchronous training waits. A slow data-loader worker, a thermal-throttled GPU, a congested network path, or a filesystem pause can reduce the throughput of the whole job. Logs should make per-rank timing visible. If only the master rank logs averaged step time, the slow rank can remain hidden until utilization is already poor.

Table 7.2 summarizes the main savings and costs of the strategies discussed in this chapter.

7.11 Operational Reliability

Large training jobs fail often enough that failure recovery is part of the design. A reliable run has a launch manifest, immutable code reference, environment capture, data manifest, checkpoint policy, health checks, and runbook. The runbook should say what to do for a single-rank failure, repeated out-of-memory errors, NaNs, slow filesystem writes, degraded network links, and validation regressions. Waiting until

failure occurs is expensive because hundreds or thousands of accelerators may sit idle while the team debates whether a checkpoint is trustworthy.

Checkpointing must match the parallel layout. Data-parallel checkpoints can be simple full replicas. FSDP and ZeRO checkpoints may be sharded. Tensor-parallel checkpoints must preserve partitioned weight layouts. Pipeline-parallel checkpoints must map layers to stages. Expert-parallel checkpoints must preserve expert identities and optimizer shards. When changing world size for resumption, the system may need a conversion step. That conversion should be tested with small models before it is needed in an emergency.

Distributed Checkpoint APIs make this layout contract explicit instead of hiding it behind `torch.save` [131]. A state-dict interface can process modules composed from FSDP, `fully_shard`, DDP, and tensor-parallel wrappers, but the checkpoint still needs a storage writer, a planner, metadata, and a load path tested under the intended topology. Asynchronous checkpointing is also not free: `async_save` separates staging completion from upload completion, and the default path may still copy data from accelerator memory to CPU memory before the background write. Newer staging helpers can move that copy to a background thread, but the runbook must still budget CPU RAM, outstanding writes, synchronization points, and what happens if training fails while an upload is in flight [130].

Data-loader reliability is a frequent bottleneck. Tokenized shards should include checksums, lengths, and format versions. The sampler should be able to resume at a token-accurate or shard-accurate position. If workers independently shuffle without a coordinated seed and epoch definition, resumption may repeat or skip data. If remote storage is used, caching policy and backpressure should be measured. A model can only train as fast as tokens arrive.

Experiment tracking should separate high-rate metrics from durable artifacts. High-rate metrics include step time, loss, gradient norm, and memory. Durable artifacts include configs, data manifests, evaluation reports, checkpoints, and environment descriptions. The first group supports live debugging; the second supports scientific interpretation after the run.

7.12 Implementation Notes

A minimal distributed training script hides many contracts. Environment variables define rank, local rank, world size, master address, and communication backend. The data sampler must partition examples so that ranks do not duplicate work unless duplication is intentional. The loss must be scaled correctly under gradient accumulation. Random seeds must differ where stochastic data order should differ and match where initialization should match. Only designated ranks should write shared artifacts, but all ranks must participate in barriers required by the checkpoint protocol.

Several checks catch serious bugs early:

1. Run one step on one GPU and on multiple GPUs with the same global batch, then compare losses within expected numerical tolerance.

2. Verify that every trainable parameter receives a finite gradient.
3. Measure peak memory by category before increasing model size.
4. Save, resume, and confirm that learning rate, token count, data position, and loss continue as expected.
5. Profile at the intended sequence length, not only at a shorter debug length.
6. Kill one worker in a controlled test if the scheduler and checkpoint system claim fault tolerance.

These tests are cheaper than discovering the same problems during a full run.

7.13 A Systems Design Checklist

A publishable systems section should expose the following quantities, because they determine whether the reported model is reproducible rather than merely impressive:

$$\mathcal{S}_{\text{train}} = (P_{\text{dp}}, P_{\text{tp}}, P_{\text{pp}}, P_{\text{ep}}, M_{\text{peak}}, C_{\text{step}}, T_{\text{io}}, R_{\text{restart}}), \tag{7.5}$$

where the P terms are data, tensor, pipeline, and expert parallel degrees, M_{peak} is peak per-rank memory, C_{step} is collective time per step, T_{io} is checkpoint or data-loading time, and R_{restart} is measured recovery time after failure. These values are not secondary engineering details: they explain why one training recipe can be repeated by another lab and another cannot.

7.14 Key Terms

All-gather A collective in which ranks exchange shards so every rank obtains the full tensor.

All-reduce A collective in which ranks receive the reduction of all contributed tensors.

Data parallelism Replicating the model across workers and averaging gradients across different data shards.

Distributed Checkpoint A checkpointing API and file format discipline for saving and loading sharded model, optimizer, and metadata state under distributed parallel layouts.

Expert parallelism Placing different MoE experts on different devices and routing token representations to them.

FSDP Fully Sharded Data Parallel, a sharded training method that keeps only local shards of model state when full parameters are not needed.

FSDP2 A newer PyTorch `fully_shard` interface that represents sharded parameters as `DTensor`s and uses hooks to materialize ordinary tensors around module computation.

Pipeline bubble Idle time caused by filling and draining pipeline-parallel stages.

Tensor parallelism Splitting individual layer tensors and computations across devices.

Tensor-parallel checkpoint A checkpoint whose tensors are saved by tensor-parallel rank and must be resumed with compatible layout metadata or merged before ordinary inference export.

Communication overlap Scheduling communication and computation so one hides behind the other instead of serializing the step.

ZeRO-Offload A ZeRO-style method that moves selected model state and optimizer work through CPU memory to reduce GPU memory pressure.

7.15 Exercises

1. Estimate persistent training memory for a 13-billion-parameter dense model using BF16 parameters, BF16 gradients, FP32 Adam moments, and FP32 master weights. Then estimate per-rank persistent memory under ZeRO-1, ZeRO-2, and ZeRO-3 for $P = 16$, ignoring activations and buffers.
2. Explain why ZeRO-3 can make a model fit in memory without reducing the arithmetic work of a single dense layer. When would tensor parallelism still be needed?
3. A model trains with 8 pipeline stages and 32 micro-batches. Estimate the simple pipeline bubble fraction. How does the fraction change with 8 micro-batches, and what other costs might increase when using 32?
4. For a pipeline-plus-expert-parallel sparse model, list the measurements needed to decide whether a DualPipe-style overlapped schedule is helping or merely hiding communication in averaged step time.
5. For an MLP block with two linear layers, describe one column-parallel and one row-parallel layout. State where all-reduce or all-gather operations are required.
6. A tensor-parallel library saves files by TP, PP, and EP rank. List the metadata needed to resume training, and explain why a separate merge step may be needed before serving the model with a standard inference stack.
7. Design a process-group layout for 128 GPUs arranged as 16 nodes with 8 GPUs per node. Choose data, tensor, and pipeline parallel degrees for a dense model and justify which collectives stay within a node.
8. A training job has stable loss but tokens per second drops by 25 percent after two hours. List at least six metrics or logs you would inspect to distinguish data loading, network, checkpointing, thermal, and straggler causes.
9. Explain why a sharded checkpoint that resumes training correctly may still be unsuitable for inference export. What metadata or conversion step is needed?
10. Design a minimal FSDP2 run card. Include device mesh dimensions, bottom-up `fully_shard` wrapping order, optimizer construction point, checkpoint API, resharding or export path, and the test that proves an ordinary model call with inputs triggers the needed all-gather hooks.
11. Compare two pipeline schedules, such as GPipe and 1F1B, for the same layer partition. What measurements would show whether the schedule improved throughput without breaking activation lifetime, optimizer-step timing, or checkpoint reconstruction?

12. Write a failure scenario for asynchronous distributed checkpointing. State how the runbook distinguishes staging completion, upload completion, durable metadata, and whether the next launch should resume from the last complete checkpoint or discard a partial write.

Chapter 8
Inference and Serving

Abstract This chapter studies large language model inference as a systems problem. It separates prefill from decode, derives KV-cache memory costs, explains batching, paged attention, chunked prefill, and prefill-decode disaggregation, compares quantization and speculative decoding, and gives practical guidance for streaming APIs, cancellation, load testing, cost accounting, and evaluation under real traffic.

Chapter contract.

The reader should leave this chapter able to explain why deployment is not just evaluation mode, estimate prefill and decode costs, reason about KV-cache residency and scheduling, and design measurements that expose latency, throughput, cost, and safety behavior under real traffic.

8.1 Serving Is Not Evaluation Mode

Inference is often introduced as "run the trained model with dropout disabled." That description is correct for a single forward pass and inadequate for a user-facing service. A large language model service must accept variable-length prompts, schedule many concurrent requests, manage KV-cache memory, stream tokens, cancel work when clients disconnect, enforce timeouts, protect shared capacity, and report cost. The same model can feel fast or unusable depending on the serving stack.

Autoregressive generation is the reason. A prompt of length T_p is processed once, but a completion of length T_g is generated one token at a time:

$$\begin{aligned} p_\theta(y_{1:T_g} \mid x_{1:T_p}) = p_\theta(y_1 \mid x_{1:T_p})\, p_\theta(y_2 \mid x_{1:T_p}, y_1) \cdots \\ \times\, p_\theta(y_{T_g} \mid x_{1:T_p}, y_{<T_g}). \end{aligned} \tag{8.1}$$

Each generated token depends on all previous tokens, so decode has a serial dependency even when each individual forward pass uses parallel hardware. Serving systems therefore optimize two different phases: prefill, which processes the prompt, and decode, which extends the sequence.

8.2 Prefill and Decode

Prefill computes hidden states for the prompt and writes keys and values for every layer into the KV cache. It resembles training inference over a full sequence: matrix multiplications are large, attention can be compute-heavy, and the system benefits from efficient attention kernels such as FlashAttention [31]. For long prompts, prefill latency can dominate time to first token. Prompt caching and prefix reuse can help when many requests share the same system prompt, retrieved context, or conversation prefix, but the cache must be keyed by exact tokens and model state.

Decode is different. At each step, the model receives the newest token, reads all previous keys and values from the KV cache, computes attention for the new position, and writes one new key/value pair per layer. The matrix multiplications are smaller, and memory bandwidth often becomes the bottleneck because the service repeatedly reads model weights and cache entries. This is why tokens per second during decode depends strongly on batch size, KV-cache layout, quantization, and attention implementation.

The distinction explains a common measurement error. Reporting only generated tokens per second hides prompt cost. Reporting only end-to-end latency hides decode throughput. A useful service dashboard separates:

Time to first token	Queueing plus prefill plus the first decode step.
Inter-token latency	The gap between streamed tokens during decode.
End-to-end latency	Time from request arrival to final token.
Input throughput	Prompt tokens processed per second.
Output throughput	Generated tokens processed per second.

These metrics move differently under load. A scheduler can improve output throughput by batching decode requests while increasing time to first token for short prompts.

8.3 The KV Cache

The KV cache stores attention keys and values for previously processed tokens. For a dense decoder with L layers, H_{kv} key/value heads, head dimension d_h, sequence length T, batch size B, and s bytes per cache element, the approximate cache memory is

$$M_{\text{KV}} = 2\, B\, T\, L\, H_{\text{kv}}\, d_h\, s. \tag{8.2}$$

The factor of two counts keys and values. For multi-head attention, H_{kv} equals the number of query heads. For multi-query attention, it may be one. For grouped-query

attention, it is between those extremes [149, 1]. This architectural choice directly affects serving capacity.

KV memory grows with active sequence length, not just model size. A smaller model with a very long context window can be harder to serve than a larger model at short context. Multi-turn chat sessions worsen the issue because every generated token extends the cache. Retrieval-augmented systems add retrieved passages to the prompt and therefore consume cache before any answer token is produced. A serving plan should budget cache memory for realistic prompt and completion length distributions, not only for a maximum context headline.

Cache precision is a systems choice. BF16 or FP16 caches preserve accuracy but consume memory. Quantized KV caches can increase batch capacity and long-context affordability, but they may affect perplexity, retrieval sensitivity, or exact copying. Any KV-cache quantization should be evaluated on tasks that depend on long-range information, not only on short generic benchmarks.

8.4 Batching and Scheduling

Batching improves hardware utilization by combining requests. In prefill, batching prompts of similar length avoids padding waste. In decode, batching many active sequences lets the service perform one larger step instead of many tiny steps. The complication is that requests arrive and finish at different times. Static batching waits to form a batch and then runs it to completion; this wastes capacity when sequences have different output lengths. Continuous batching admits new requests as old ones finish, keeping the decode batch full.

Continuous batching turns serving into a scheduling problem. Each request has prompt length, generated length so far, maximum token budget, priority, deadline, and memory footprint. The scheduler chooses which requests enter prefill, which active sequences decode next, and when to evict or pause work. Policies often balance throughput against latency. A throughput-optimized policy may batch aggressively and hurt short interactive requests. A latency-optimized policy may keep batches small and waste hardware.

Toy continuous-batching implementations are useful because they expose the state that production systems must make explicit. A request manager tracks an id, prompt tokens, generated tokens, current length, maximum length, and a waiting/running/completed state; a KV-cache manager tracks fixed or paged slots, per-slot sequence lengths, a request-to-slot map, a slot-to-request map, and the free-slot set. The prefill path allocates slots, pads prompts only for the model batch, writes the prompt KV entries, and returns logits for the first generated token. The decode path feeds only the last generated token for each active slot together with its current position, writes one KV entry per layer, updates length, tests EOS or length stop conditions, and frees the slot immediately when the request finishes. This is the operational meaning of continuous batching; without these state transitions, the phrase hides the hard part.

Several details that are harmless in a notebook become service contracts. Request queues must be thread-safe if producers and the inference loop run concurrently. Max-

imum batch size and maximum sequence length should be derived from a memory budget, not chosen independently. A fixed tensor cache of shape layers by batch by sequence by heads by head dimension is easy to teach, but it over-reserves memory for short requests and fragments capacity for mixed traffic; a paged cache changes the allocation granularity but still needs the same request-state bookkeeping. If a decode step finishes a request, the service must remove it from the active batch before the next step and must make the cache slot reusable without exposing stale tokens across tenants.

PagedAttention, introduced in vLLM, addresses memory management by storing KV cache in fixed-size blocks rather than requiring each sequence to occupy a contiguous allocation [77]. The analogy to virtual memory is useful: logical token positions map to physical cache blocks, and blocks can be shared, allocated, or freed as sequences grow and finish. This reduces fragmentation and makes continuous batching more effective. It does not remove the KV-cache cost; it makes the cost manageable under variable-length traffic.

Recent serving stacks also separate scheduling by phase. Prefill is usually compute-heavy and benefits from large prompt batches; decode is often memory-bandwidth-bound and benefits from keeping many active sequences resident. Chunked prefill splits a long prompt into smaller pieces so it does not monopolize the accelerator while short decode steps wait. Prefill-decode disaggregation goes further by placing prefill and decode on different workers or hardware pools, then transferring the resulting KV state or a system-specific representation between them [42]. This can improve mixed-traffic latency, but it adds routing, cache-transfer, admission-control, and failure-recovery contracts.

The phase goals should be written explicitly in a production design. A prefill pool is optimized for input tokens per second, prompt-batch utilization, and time to first token; a decode pool is optimized for output tokens per second, inter-token latency, KV-cache residency, and memory bandwidth. The handoff between them is a first-class path: it must preserve the exact token prefix and positional state, account for KV transfer time, and expose backpressure when decode workers cannot accept more resident sequences. For a DeepSeek-V3-like MoE service, the report should also say whether MLA-style cache compression, expert parallel routing, FP8 or other inference precision, and MTP heads are actually active at serving time rather than only appearing in the training recipe [34].

MoE serving adds another layer to the same scheduler. Expert routing can create all-to-all communication during inference, and the preferred placement for experts may differ from the placement for attention or dense FFN work. A service report for a sparse model should therefore separate prefill throughput, decode throughput, expert token counts, all-to-all time, cache movement, and tail latency. Otherwise an apparent batching improvement may simply hide expert-routing congestion.

Figure 8.1 shows the serving loop as a resource-control system around prefill, decode, cache blocks, filters, and streaming.

Table 8.1 makes the traffic-shape dependence explicit.

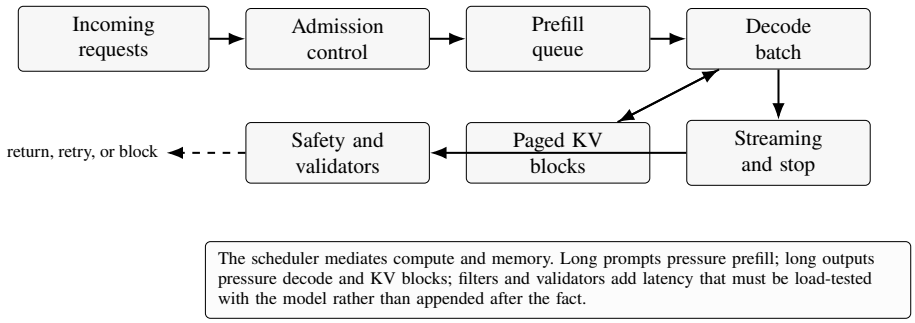

Fig. 8.1 An inference service as a resource-control system. The PagedAttention idea from vLLM motivates the KV-block abstraction, but the figure is an original synthesis of the full serving loop [77].

Table 8.1 Serving bottlenecks differ across request shapes.

Traffic Pattern	Likely Bottleneck	Useful Metric
Short prompts, short outputs	Queueing and launch overhead	Requests/s, p95 end-to-end latency
Long prompts, short outputs	Prefill compute and attention memory	Time to first token, input tokens/s
Short prompts, long outputs	Decode bandwidth and batch occupancy	Inter-token latency, output tokens/s
Long conversations	KV-cache capacity	Active tokens, cache hit/eviction rates
Shared system prompts	Prefix-cache management	Cache hit rate, first-token latency by prefix
Mixed priorities	Scheduler policy	Latency by class, deadline miss rate

8.5 Memory Management and Admission Control

An inference server needs an admission policy. If it accepts more active tokens than the hardware can support, it will either run out of memory or degrade latency for every request. A simple policy estimates each request's maximum cache reservation from prompt length plus maximum new tokens. A more flexible policy admits based on expected length and enforces a hard cap during generation. The latter improves utilization but needs cancellation and partial-progress handling when a request exceeds budget.

Memory fragmentation is not a theoretical concern. Without paging or careful pooling, variable-length sequences allocate and free differently sized cache regions, leaving holes that cannot satisfy later long requests. Fragmentation can produce out-of-memory errors even when total free memory appears large. Fixed block allocation, request compaction, and preallocated cache arenas are common mitigations.

Eviction policy depends on product semantics. For stateless completion APIs, the cache can be freed when the response ends. For chat sessions, a server may keep conversation prefixes warm for a short time. For retrieval systems, retrieved documents may change between turns, so prefix reuse must respect the exact token sequence and

tool outputs. For privacy-sensitive deployments, cache lifetime and tenant isolation are security concerns as well as performance concerns.

8.6 Quantization and Compression

Inference cost is often dominated by memory bandwidth and capacity. Quantization reduces the bytes moved for weights, activations, or KV cache. Weight-only quantization stores weights in low precision while computing with higher-precision activations. This can reduce model memory and improve decode throughput, especially when batch sizes are small and weight loading dominates. GPTQ and AWQ are influential post-training approaches for low-bit LLM weight quantization [43, 95]. SmoothQuant targets weight-and-activation INT8 quantization by moving quantization difficulty from activations to weights through offline scaling [178].

Quantization is not a single knob. Important choices include bit width, per-tensor versus per-channel scaling, group size, calibration data, outlier handling, kernel support, and whether the target hardware accelerates the chosen format. A 4-bit checkpoint without efficient kernels may save memory but fail to improve latency. A calibration set that omits code, math, multilingual text, or long-context examples may hide severe regressions.

Evaluation should include both quality and systems metrics. Quality metrics include held-out perplexity, task accuracy, exact-match behavior for copying, code execution, long-context retrieval, and safety probes. Systems metrics include model load time, peak memory, time to first token, decode tokens per second, batch capacity, and cost per million tokens. Quantization can improve one metric while hurting another; a publishable claim states the traffic shape and hardware.

8.7 Speculative and Parallel Decoding

The serial dependency of autoregressive decoding makes acceleration difficult. Speculative decoding uses a smaller or cheaper draft model to propose several tokens, then verifies those tokens with the target model in a more parallel computation. The exact algorithm can preserve the target model's output distribution while reducing latency when the draft model is accurate enough and cheap enough [84]. The speedup is not automatic: if draft tokens are frequently rejected, the service pays for both draft and target work.

Speculative decoding introduces new serving decisions. The draft model must be deployed, scheduled, and versioned with the target. The acceptance rate should be monitored by prompt class, language, temperature, and domain. Greedy decoding, nucleus sampling, and temperature sampling have different verification details. A model that is excellent for English chat may be a poor drafter for code or low-resource languages. The operational metric is not only accepted tokens per target pass, but end-to-end latency after adding draft-model overhead.

Other parallel decoding methods add auxiliary heads or predict multiple future tokens directly. These approaches can reduce the number of target passes but may require model modification or extra training. They should be evaluated under the same rule as speculative decoding: measure quality-preserving speedup on the actual request distribution, including queueing and memory pressure.

Multi-token prediction needs careful terminology. In a pretraining report, an MTP head may be an auxiliary loss that improves shared representations and is removed after training, as in the DeepSeek-V3 recipe [34]. In a serving system, a multi-token head or draft path must remain available at inference time and must preserve output quality under the decoding policy. These are different claims. A service report should state whether future-token prediction affects the released checkpoint, the runtime decoder, or both.

The implementation also determines what "multi-token" means. A simple auxiliary head can directly predict token $t + i$ from the current hidden state, but a DeepSeek-V3-style MTP module is closer to a recurrent next-token-prediction stack: later auxiliary modules consume shifted embeddings and latent features, share parts of the embedding or language-head path, and train with next-token losses at shifted positions. During inference, any retained MTP path must still manage its own cache, acceptance rule, and rollback behavior. Calling both designs MTP is acceptable only if the report distinguishes training loss, checkpoint structure, and runtime decoding algorithm.

8.8 Attention Kernels and Long Context

Attention kernels matter in prefill and long-context serving. Standard attention materializes or implicitly handles an attention matrix whose logical size grows with T^2. FlashAttention improves memory efficiency by tiling computation and avoiding unnecessary reads and writes to high-bandwidth memory while computing exact attention [31]. The key invariant is online softmax: as a query streams over key/value blocks, the kernel maintains a running row maximum and normalizer, rescales partial outputs when a later block changes the maximum, and therefore avoids materializing the full score matrix without changing the mathematical attention result. This is especially important for long prompts, training, and batched prefill. During single-token decode, the attention pattern is a query attending to cached keys and values, so the bottleneck often shifts toward cache bandwidth and layout.

The acceleration methods in modern serving stacks are best grouped by what bottleneck they attack. Architectural changes such as MQA and GQA reduce KV heads and therefore decode bandwidth. Kernel changes such as FlashAttention-2 improve tiling, work partitioning, and occupancy for prefill and training-like attention [30]. Cache-management systems such as PagedAttention reduce fragmentation rather than changing model math. Long-context variants such as sliding-window attention, StreamingLLM attention sinks, and LongLoRA's shifted sparse attention change which tokens participate in training or serving computation [179, 23]. Decode accelerators such as speculative decoding, Medusa, and Lookahead decoding try to reduce the number of serial target-model steps [84, 16, 44]. These methods compose only when their contracts

Table 8.2 A serving-oriented taxonomy of attention and decoding accelerators.

Family	Examples	Primary bottleneck	Contract to report
KV-head reduction	MQA, GQA	Decode cache size and bandwidth	KV heads, cache bytes/token, quality change
Kernel and tiling	FlashAttention, FlashAttention-2	Prefill and long-sequence attention I/O	Exactness, dtype, hardware, sequence lengths
Cache management	PagedAttention	Fragmentation and continuous batching	Block size, eviction, sharing, cancellation
Windowed or sink attention	Sliding window, StreamingLLM, LongLoRA	Long-context memory and training cost	Which tokens remain visible, whether inference uses full attention
Parallel decoding	Speculative decoding, Medusa, Lookahead	Serial target-model steps	Acceptance rate, extra heads or draft model, quality preservation

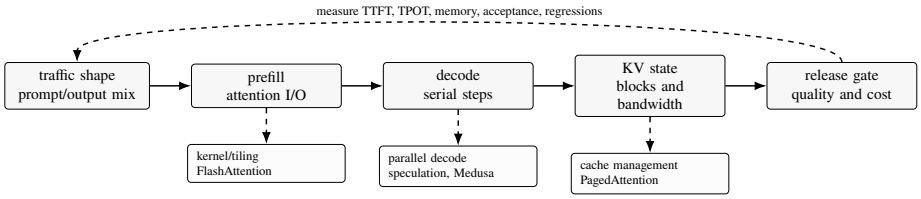

Fig. 8.2 A serving accelerator should be chosen by bottleneck, not by name recognition. The map synthesizes kernel, cache-management, and parallel-decoding contracts from FlashAttention, PagedAttention/vLLM, speculative decoding, Medusa, and Lookahead decoding [31, 77, 84, 16, 44].

match: a long-context attention trick can help prefill while doing little for decode; a multi-head decoder can reduce target passes while increasing checkpoint and routing complexity.

Table 8.2 groups these methods by the bottleneck they primarily attack.

Figure 8.2 turns the same taxonomy into a serving design map: a useful accelerator is one whose contract matches the dominant bottleneck in the current traffic mix.

Long context should be evaluated with length-aware tests. A model can accept a long prompt syntactically yet fail to use evidence in the middle. A serving stack can support a maximum context in isolation yet collapse under many concurrent long-context requests. A benchmark that reports only accuracy at maximum length omits the cost curve. For production planning, the relevant graph plots latency, throughput, memory, and accuracy as functions of input length and active batch size.

Chunked prefill is a practical technique for long prompts. Instead of monopolizing the device with one long prefill, the server splits prompt processing into chunks and interleaves other work. This can reduce tail latency for mixed traffic, but it complicates scheduling and may reduce raw throughput. Prefix caching, retrieval caching, and chunked prefill should be measured together because they interact.

8.9 Streaming APIs, Cancellation, and Safety Filters

Streaming changes user experience and server behavior. A streamed response can show the first token quickly while the full answer is still decoding. The server must flush partial text, handle token-to-text boundaries, and decide when to emit tool-call or structured-output fragments. It must also handle cancellation. If a client disconnects, continuing to decode wastes capacity and may hold KV-cache blocks that other requests need.

Timeout policy should distinguish queue timeout, prefill timeout, decode timeout, and total response timeout. A request that waits too long in the queue has not consumed much GPU work; a request that times out during a long decode may already have consumed substantial compute. Retrying blindly can multiply load during incidents. Backpressure, rate limits, and clear error semantics are part of serving correctness.

Safety filters and output validators add latency. Some systems filter prompts before prefill, monitor generated tokens during streaming, and run final classifiers after completion. Tool-using systems may validate JSON schemas or execute sandboxed checks. These components should be in the latency budget and load test, not treated as free post-processing. If a safety classifier uses the same GPU pool as generation, it competes for capacity.

8.10 Load Testing and Cost Accounting

A load test should model traffic, not merely saturate a server with identical prompts. Real traffic has distributions over prompt length, output length, decoding parameters, tenant priority, cache reuse, and cancellation. Synthetic tests are still useful, but they should state the distribution. A minimal load-test report includes average, p50, p90, p95, and p99 latency; time to first token; inter-token latency; input and output tokens per second; requests per second; error rate; cancellation rate; GPU memory; GPU utilization; and cost per million input and output tokens.

Cost per token is not constant. Long prompts spend more prefill compute. Long outputs spend more serial decode time and KV memory. Small batches may waste hardware. Very large batches may improve throughput while hurting latency. Quantization can reduce cost if kernels are efficient. Speculative decoding can reduce cost only if the draft overhead is smaller than the target passes it avoids. A pricing or capacity model should therefore separate input-token and output-token economics.

Tail latency deserves special attention. Users experience the slow request, not the mean. Tail latency can come from queueing, long prompts, cache fragmentation, straggler kernels, network backpressure, safety filters, or retries. Averages hide all of these. A good load test captures traces for the slowest requests and attributes time to queue, prefill, decode, post-processing, and streaming flush.

Table 8.3 A compact load-test table for a large language model service.

Scenario	TTFT p95	E2E p95	Output toks/s	Cost / 1M toks
Short chat				
Long prompt QA				
Code generation				
RAG with shared prefix				
Mixed production trace				

8.11 Worked Capacity Example

Consider a 32-layer model with $H_{\text{kv}} = 8$, $d_h = 128$, BF16 cache elements, and an active cache budget of 40 GiB after weights and runtime buffers. Each token in one sequence consumes approximately

$$2 \times L \times H_{\text{kv}} \times d_h \times 2 = 2 \times 32 \times 8 \times 128 \times 2 = 131{,}072 \text{ bytes} \tag{8.3}$$

of KV storage. A 40 GiB cache contains about 42,949,672,960 bytes, so it can hold roughly $42{,}949{,}672{,}960/131{,}072 \approx 327{,}680$ active sequence tokens before fragmentation and block overhead. That budget could support about 80 conversations at 4096 active tokens, or 20 long-document requests at 16k active tokens, but not both at the same latency target. This arithmetic is crude, yet it is the kind of calculation that should precede claims about long-context deployment.

8.12 Implementation Notes

A minimal generation loop is easy to write and hard to serve. The loop tokenizes a prompt, runs prefill, samples a token, appends it, and repeats until an end condition. A service-grade loop separates request state from model execution. Request state includes token ids, sampling parameters, stop sequences, maximum token budget, KV-block table, stream handle, priority, deadline, and cancellation flag. Model execution operates on batches selected by the scheduler.

Sampling code must be deterministic when determinism is promised. Temperature, top-k, top-p, repetition penalties, logit biases, and random seeds should be applied in a defined order. Stop sequences are text-level or token-level depending on the API; confusing the two can leak partial delimiters or stop too late. Structured-output modes need validators that can operate incrementally or repair invalid partial generations.

Small inference demos also reveal common reproducibility gaps. If the runtime prompt template differs from the SFT template, the report should say so rather than attributing the result only to the model. If a LoRA adapter ships with an expanded tokenizer, the base embedding and LM-head shapes may need resizing before generation. If an NTK or RoPE scaling patch is applied at load time, the alpha value and maximum

context assumption are part of the serving configuration. Streaming wrappers often receive cumulative token ids and then slice off the prompt; they must handle EOS, partial token-to-text boundaries, client cancellation, and cache cleanup. Character-count history truncation is not a substitute for token-budgeted context management because it can cut across templates, roles, or special tokens.

The service-grade invariants are stable across implementations: separate prefill and decode metrics, budget KV cache explicitly, test variable-length batching, account for cancellation, and evaluate quality after every compression or decoding optimization.

8.13 Key Terms

Continuous batching A scheduling policy that adds and removes requests from active decode batches as they arrive and finish.

Chunked prefill Splitting long prompt processing into smaller chunks so prefill work can be interleaved with decode and other requests.

Decode The autoregressive phase that generates one or more new tokens using the KV cache.

Disaggregated serving A serving design that separates phases or components such as prefill and decode across different workers or hardware pools.

KV cache Stored attention keys and values for previous tokens, used to avoid recomputing the full prefix at every decode step.

Medusa A parallel decoding method that adds extra heads to propose multiple future tokens and verifies candidate continuations with the base model.

PagedAttention A KV-cache management method that stores cache entries in fixed-size blocks to reduce fragmentation under variable-length requests.

Prefill The phase that processes prompt tokens and initializes the KV cache before generation.

Speculative decoding A method that proposes tokens with a cheaper draft computation and verifies them with the target model.

StreamingLLM A windowed KV-cache method that preserves early attention-sink tokens together with recent tokens for streaming long inputs.

Time to first token The latency from request arrival until the first output token is available to stream.

8.14 Exercises

1. Derive the KV-cache memory for a model with $L = 32$, $H_{\text{kv}} = 8$, $d_h = 128$, BF16 cache elements, batch size $B = 24$, and active sequence length $T = 4096$. How does the answer change under multi-query attention with $H_{\text{kv}} = 1$?
2. Design a scheduler for mixed traffic containing short chat, long-document summarization, and code generation. State the admission rule, batching rule, and cancellation rule.

3. Build a load-test table using Table 8.3. Add p50 and p99 columns, input tokens/s, error rate, and cache hit rate. Explain which metric would first reveal KV-cache fragmentation.
4. A 4-bit weight-only quantized model has lower memory use but worse p95 latency than BF16. List five possible causes and one measurement that would distinguish each cause.
5. Explain speculative decoding with a draft model and a target model. Under what prompt distributions would you expect low acceptance rates?
6. Compare two deployments of the same model: one optimized for maximum throughput and one optimized for interactive latency. Specify batch size policy, timeout policy, and the metric each deployment should optimize.
7. Design a prefill-decode disaggregated serving plan for mixed traffic with long RAG prompts and short chat completions. State what state must move between workers and which metric would reveal that disaggregation made tail latency worse.

Part III
Adaptation

Chapter 9
Supervised Instruction Tuning

Abstract This chapter explains how pretrained models become instruction-following systems through supervised examples. It covers chat templates, task diversity, Alpaca-style self-instruction, synthetic data, refusal data, data quality, task-specific SFT, loss masking, and the limits of imitation before preference learning or deployment evaluation.

Chapter contract.

The reader should leave this chapter able to convert a base-model training view into an assistant-interface training view, specify chat-template and label-mask contracts, evaluate instruction data quality, and explain why imitation is necessary but insufficient for alignment.

9.1 The Interface Shift

A pretrained decoder is trained to continue text. A deployed assistant is expected to answer a request, respect a role hierarchy, follow a formatting contract, refuse some requests, use tools when permitted, and stop when the answer is complete. Supervised instruction tuning, usually shortened to SFT, is the first post-training stage that teaches this interface. It does not change the causal language-model factorization from Chapter 1; it changes the distribution of sequences on which the model is optimized.

Let an instruction example contain a context c and a target assistant response a. The context may include a system message, one or more user turns, previous assistant turns, retrieved passages, tool observations, or task-specific fields. A chat template τ turns the structured example into a token sequence

$$z_{1:T} = \tau(c, a), \tag{9.1}$$

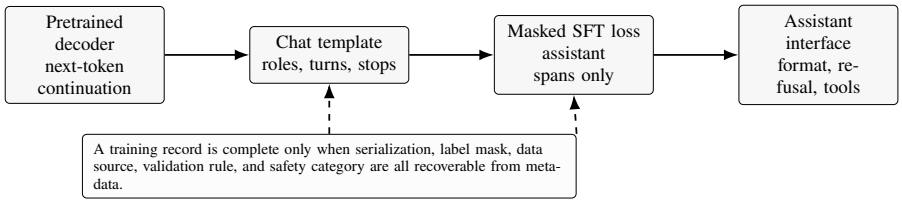

Fig. 9.1 SFT changes the interface distribution rather than the causal language-model factorization. The engineering contract is the combination of template, masking, metadata, and validation, not merely a prompt-answer pair.

where some tokens are input-only and other tokens are response targets. SFT minimizes next-token loss only on the target positions,

$$\begin{aligned} M_{\text{mask}} &= m_1 + \cdots + m_T, \\ g_t &= m_t \log p_\theta(z_t \mid z_{<t}), \\ \mathcal{L}_{\text{SFT}}(\theta) &= -(g_1 + \cdots + g_T)/M_{\text{mask}}. \end{aligned} \tag{9.2}$$

where $m_t = 1$ for response tokens to be imitated and $m_t = 0$ for prompt tokens, padding, and usually separator tokens. In many training libraries this mask is implemented by setting ignored labels to `-100`. The small detail matters: if the loss is charged on user instructions or system text, the model is trained to generate prompts rather than only answers.

Instruction tuning became central because large base models already possess many latent capabilities, but they do not reliably infer the desired user-facing protocol from a raw prompt. FLAN-style work showed that fine-tuning on tasks expressed as natural-language instructions improves zero-shot generalization to unseen task types [173]. InstructGPT used supervised demonstrations as the first stage before reward modeling and RLHF, making clear that SFT is useful but not a complete alignment method [125]. Modern open model reports treat SFT as one component of a larger post-training pipeline with preference data, safety data, red-teaming, and deployment evaluation [166, 48].

Instruction tuning and downstream task SFT should be distinguished in reports. Instruction tuning uses broad, reusable examples whose instruction text describes the task in natural language, so the model learns a general interface across translation, summarization, extraction, coding, dialogue, and refusal. Task SFT is narrower: it adapts a model to a product workflow, domain corpus, schema, or customer style, and it may be implemented as full fine-tuning or as a PEFT run. The same loss can be used in both cases, but the evidence required is different. Instruction tuning should prove breadth and held-out task generalization; task SFT should prove workflow improvement without losing base capabilities.

Figure 9.1 summarizes this shift from continuation modeling to a serialized assistant interface.

Table 9.1 Common SFT data types and the contracts they teach.

Data Type	Contract Being Taught	Typical Failure If Under-Specified
Single-turn instructions	Map one user request to one answer	Verbose, generic, or format-breaking answers
Multi-turn dialogue	Maintain state, resolve references, and repair mistakes	Forgetting earlier constraints or over-trusting stale context
Structured tasks	Emit JSON, SQL, code, labels, or other constrained outputs	Invalid syntax despite semantically plausible text
Tool traces	Decide when to call a tool and how to use observations	Hallucinated tool results or repeated invalid calls
Refusal and safety data	Decline disallowed requests while helping with safe alternatives	Over-refusal, under-refusal, or inconsistent tone
Domain demonstrations	Adopt domain vocabulary and workflow conventions	Shallow style imitation without factual grounding

9.2 Instruction Data as a Contract

An instruction dataset is not merely a list of prompts and answers. It is a behavioral contract for the model. A good record states who is speaking, what authority each message has, which external evidence is available, what output format is required, and what should happen when the request is impossible or unsafe. A weak record leaves these details implicit, so the model learns correlations that are convenient in the dataset but fragile in deployment.

Table 9.1 summarizes common SFT example types. A single training run often mixes them, but the mixture should be explicit. The model should not learn that every user request deserves a long explanation, that every code question requires a full file rewrite, or that every safety-sensitive request is either always allowed or always refused. Diversity is useful only when it is organized.

The template is part of the dataset. For a chat model, the same semantic example can train different behavior depending on whether the text is formatted as `system/user/assistant` messages, a plain prompt-completion pair, or a vendor-specific instruction format. The template should make message boundaries unambiguous, include an end-of-turn token or stop condition, and match the inference template used in serving. A mismatch between training and inference templates is one of the easiest ways to make a capable checkpoint appear broken.

Data quality must be evaluated at several levels. At the example level, the answer should be correct, complete, and consistent with the instruction. At the dataset level, tasks should be balanced across skills, lengths, languages, domains, and refusal categories. At the provenance level, the builder must know which sources can be redistributed, which sources may contain personally identifiable information, and which benchmarks must be excluded from training. Benchmark contamination is especially serious in instruction tuning because many public evaluation tasks are also phrased as instructions [153, 180].

9.3 Synthetic Instruction Data

Human-written demonstrations are expensive. Synthetic instruction generation tries to expand coverage by asking a capable model to create new tasks, inputs, and answers. Self-Instruct is the canonical recipe: seed the process with a small set of human-written instructions, prompt a model to generate new instructions and demonstrations, filter malformed or near-duplicate examples, and then fine-tune on the resulting set [172]. Alpaca-style recipes popularized the same idea for open models by distilling instruction-following behavior from stronger teacher models.

The original Self-Instruct pipeline is useful because it makes the data contract concrete. It starts from 175 seed tasks, repeatedly samples seed and generated instructions as in-context examples, asks the model for new instructions, classifies whether a task is a classification task, and then generates instances using input-first or output-first templates. The filtering step removes near-duplicate instructions by similarity, malformed generations, overly short or long instructions, and inconsistent input-output pairs. The released run produced roughly 52K instructions and 82K instances after filtering. A reproduction should therefore report the seed set, in-context sampling mix, task-type classifier prompt, instance-generation template, similarity threshold, invalid-generation heuristics, and the final instruction/instance counts.

Synthetic data is attractive because it can cover many formats quickly. It can create paraphrases, edge cases, schema-specific examples, tool traces, or domain-style dialogues long before a human annotation program has reached the same breadth. It is also dangerous because errors scale with the generator. A teacher that fabricates facts, mishandles code, writes insecure SQL, or refuses too aggressively can stamp those mistakes into thousands of training examples. Filtering must therefore test both surface diversity and semantic validity.

Modern synthetic-data pipelines are more than prompt expansion. Evol-Instruct-style methods deliberately rewrite seed tasks into harder variants [181]; Orca-style distillation trains smaller models from richer teacher traces rather than only final answers [114]; STaR-style bootstrapping keeps generated rationales that lead to correct answers and repeats the loop [194]. Rejection sampling is the common control mechanism: sample many candidates, score them with unit tests, exact answers, retrieval support, reward models, or human audits, then train only on accepted examples. This turns data generation into a search problem, and it should be logged with the same care as model training: generator version, prompt template, sampling parameters, acceptance rules, rejection reasons, and overlap checks against benchmarks.

One useful formalization is to treat synthetic data as a filtered proposal distribution. A teacher or generator samples candidates $e \sim q_\omega(e \mid s, \pi)$ from seed examples s and a generation policy π. Validators $v_1, \ldots, v_K$ then define an accepted set

$$\mathcal{D}_{\text{accept}} = \{e : v_k(e) = 1 \text{ for all required gates } k\}.$$

The important modeling choice is not only q_ω but the gates. A code example without executable tests, a math example without a checked answer, or a safety example without near-neighbor contrast is weak supervision even if it is fluent.

Distillation also changes what is being learned. A student can imitate the teacher's surface style without acquiring the teacher's latent competence; it can inherit the teacher's safety boundary; or it can become brittle because the accepted data contains only easy successes. For reasoning and tool use, the strongest synthetic records usually include the problem, context, tool observations or verifier results, the final answer, and a compact explanation of why the answer is valid. For safety-sensitive domains, teacher-generated examples require additional review because fluency can hide unsupported medical, legal, financial, or security claims.

A practical synthetic-data pipeline usually has four gates. First, generation prompts should specify the task family, difficulty, allowed sources, answer format, and exclusion rules. Second, automatic filters should remove examples that are too short, too similar, malformed, unsupported by evidence, or outside policy. Third, task-specific validators should execute or check outputs when possible: parse JSON, run unit tests, execute SQL against a sandbox database, or compare against a known answer. Fourth, a sampled human audit should estimate residual error rates and update the generator prompts. The goal is not to make synthetic data look diverse; it is to make it add reliable coverage that the current model lacks.

Small curated datasets can be more valuable than broad synthetic volume. Platypus-style tuning used a small Open-Platypus mixture focused on STEM and logic, emphasized human-designed questions, removed exact duplicates and high-similarity instructions, and checked training questions against benchmark questions before LoRA tuning and merging [79]. The lesson is not that every model should become a STEM specialist. It is that SFT data reports should include source-level counts, license and redistribution status, domain filters, duplicate-removal method, benchmark-contamination checks, and whether merged adapters were themselves trained on auditable data.

Language-specific Alpaca variants add another reporting obligation. Chinese Alpaca used Chinese LLaMA as the base, combined millions of instruction examples from translation, public Chinese tasks, Alpaca-style data, and crawled or teacher-generated SFT records, and expanded the Plus versions with more STEM and scientific data [29]. Its template choice is a reminder that "Alpaca format" is not a single invariant: the reported setup used the no-input Alpaca prompt template and concatenated instruction and input fields when needed. A multilingual SFT report should therefore state template variants, language proportions, whether data was translated or originally written in the target language, maximum sequence length, dynamic padding behavior, and whether benchmark-like public tasks were excluded.

Synthetic data also changes the ethics and licensing story. A generated answer can still be a derivative of the teacher model's training distribution, a reproduction of memorized text, or an unsafe medical or legal recommendation. The dataset builder must document the teacher model, prompts, filters, and known failure modes. If those details are absent, downstream users cannot interpret the resulting checkpoint.

9.4 Training Details

The training loop for SFT looks simple: tokenize examples, form batches, run the causal model, compute masked cross-entropy, and update parameters. Most failures come from the details around that loop.

First, label masks should align with message boundaries. In a single-turn example, user tokens normally condition the answer but do not contribute to the loss. In a multi-turn example, previous assistant turns may be treated either as context only or as additional target spans. Both choices are defensible, but they train different behavior. If previous assistant turns are targets, the model receives more supervised tokens per dialogue; if they are context only, the objective focuses on the final answer. The choice should be recorded.

Second, the model must be supervised to stop. If the assistant span does not include an end-of-turn or EOS target, or if the label mask accidentally ignores that token, the model can learn fluent answers without learning a boundary. The deployment symptom is "infinite" generation until the server hits `max_new_tokens`. Stop behavior should be tested with the same chat template, special tokens, and stop sequences used in serving.

Third, sequence packing improves accelerator utilization but can silently corrupt examples. Packing concatenates multiple short conversations into one fixed-length sequence. The attention mask and position handling must prevent a response in one packed example from attending to the user request in the next example unless the packing format deliberately inserts a boundary that the model can learn to ignore. If the implementation uses ordinary causal attention across packed examples, it may train on artificial cross-example context.

Completion-only SFT collators make this even more concrete. A common implementation searches the tokenized sample for a response delimiter such as `###Answer:` and masks everything before it. That is correct only if the delimiter token ids are tested with the exact tokenizer, leading newline, spacing, and special-token settings used in preprocessing. If a packed dataset path and a completion-only collator path coexist, the run report should say which one was used: packing improves utilization, while delimiter-based masking gives cleaner assistant-only supervision, and combining the two requires explicit boundary tests.

Hand-built chat templates expose the same problem. If a preprocessing script first inserts textual markers such as `<EOS>` or a custom label-start token and only then calls the tokenizer, the marker may be split, normalized, or ignored instead of becoming the intended single id. A safer educational pattern is to tokenize each role prefix and content span explicitly, concatenate token ids, and build an `is_label` vector at the same time. The collator can then convert non-label positions to the ignore index and shift labels according to the framework's convention. A one-example unit test should print the decoded sequence, the special-token ids, the assistant mask, and the final labels; otherwise an SFT run can silently train on the wrong span or omit the EOS target.

Fourth, padding is not a cosmetic choice. Right padding is common during training because it keeps examples in natural order, but batched decoder-only generation often uses left padding so the next-token position aligns with the last non-padding token. Either choice can work if attention masks, position ids, labels, and KV-cache initialization are

consistent. A mismatch can appear as wrong continuation positions, bad loss masks, or broken batch inference even when single-example generation looks fine.

Fifth, length control is part of the objective. Truncating from the right may remove the target answer; truncating from the left may remove the instruction; truncating retrieved evidence may make a correct answer appear unsupported. A robust preprocessing job reports the fraction of examples truncated, the number of target tokens lost, the distribution of prompt and answer lengths, and the examples most affected.

Sixth, batch composition affects the gradient. A mixture with many long coding answers may dominate token-weighted loss even if those examples are a minority by count. Conversely, many short classification examples may dominate example-weighted sampling while contributing few target tokens. A useful run log reports both example counts and target-token counts by data source.

The optimizer settings are usually less exotic than pretraining but still important. SFT often uses a lower learning rate than training from scratch, short warmup, weight decay on non-norm parameters, gradient clipping, and early stopping by held-out loss or task metrics. Parameter-efficient variants, especially LoRA and QLoRA, are common when full fine-tuning is too expensive; Chapter 10 studies their tradeoffs. PEFT may need more update steps or more passes over the data than full fine-tuning because the trainable subspace is smaller, so reports should compare target tokens, validation curves, and regression behavior rather than only epoch counts. Full fine-tuning can alter the model more deeply, but it also stores a full adapted checkpoint and increases the risk of catastrophic forgetting when the SFT set is narrow.

9.5 Refusal and Safety Data

Instruction tuning can teach refusals, but refusal data must be designed with care. The easy mistake is to train a binary style: answer normal requests, refuse unsafe requests. Real deployments require finer distinctions. A model should refuse requests for wrongdoing, protected private data, or dangerous operational details; it should answer benign nearby requests; it should ask clarifying questions when risk depends on context; and it should avoid turning every refusal into a lecture.

Safety examples therefore need paired contrastive coverage. For instance, a dataset might include a disallowed request for evading an access control system, an allowed request to explain access-control principles, and an allowed request to audit a user's own configuration. Without such near-neighbor examples, the model may learn surface keywords rather than policy boundaries. This is one reason SFT is usually followed by preference optimization and red-team evaluation rather than treated as the final safety layer [125, 166].

Refusal data also interacts with domain adaptation. In medicine, law, finance, education, and internal enterprise settings, a helpful response may require caution, uncertainty, and referral rather than a simple answer or refusal. A model trained only on generic safety data can sound polished while giving advice outside its evidence base. Chapter 11 returns to this problem for specialized domains.

9.6 Evaluation

SFT evaluation should answer three questions: whether the model follows instructions, whether it preserves base capabilities, and whether it behaves safely under distribution shift. A single benchmark score cannot answer all three.

Instruction following can be measured with held-out task suites, format validators, human pairwise comparisons, and model-graded rubrics. Format-heavy tasks should use deterministic checks whenever possible: a JSON parser is more reliable than a preference judge for JSON validity, and SQL execution is more reliable than surface matching when queries are semantically equivalent. Free-form tasks need human or high-quality rubric-based review, but the review prompt must not leak the desired answer.

Capability preservation requires regression tests against the base model. SFT can improve chat behavior while degrading factual recall, translation, multilingual quality, code generation, or calibration. This is especially common when the SFT set is narrow, overly synthetic, or dominated by one language. A serious report includes before-and-after scores on tasks that were not optimized, not only a leaderboard where the tuned model looks good.

Safety evaluation must include both direct unsafe requests and benign requests that share vocabulary with unsafe requests. It should measure under-refusal and over-refusal separately. A model that refuses everything in a sensitive domain is not safe in a user-facing sense; it is unusable. A model that answers every request politely is not safe either. The operational target is a calibrated policy boundary with useful safe completions.

Finally, evaluation data needs quarantine. If public benchmark examples are used during SFT, then later benchmark scores are not evidence of generalization. Even when the exact examples are removed, paraphrases and synthetic variants can leak. The evaluation set should have documented provenance, a creation date, and a contamination check against the training mixture.

9.7 Limits of Imitation

SFT is imitation learning. It teaches the model to place probability mass on responses that look like demonstrations. It does not directly teach which of two plausible responses humans prefer, how to trade off helpfulness and safety, whether a retrieved passage is trustworthy, or how to recover after a tool fails. If the demonstrations contain confident hallucinations, the model learns confident hallucinations. If the demonstrations are verbose, the model becomes verbose. If the demonstrations always comply, the model learns compliance even where refusal is appropriate.

This limitation does not make SFT optional. SFT establishes the interface on which later methods operate. Preference learning needs candidate assistant responses to compare; tool-use training needs a call format; retrieval systems need answer synthesis behavior; safety tuning needs a refusal style. The right conclusion is that SFT should be narrow in its claims and disciplined in its data. It is the stage that turns a base model

into a model that can participate in an assistant protocol, not the stage that proves the assistant is reliable.

9.8 Key Terms

Chat template A deterministic serialization of structured messages, roles, separators, and stop tokens into the token sequence consumed by the model.

Instruction tuning Supervised fine-tuning on examples that express tasks as natural-language requests and desired assistant responses.

Label mask A binary or ignored-label pattern specifying which positions contribute to the SFT loss.

Self-instruction A synthetic-data method in which a model generates new instructions and demonstrations that are filtered and used for fine-tuning.

Benchmark contamination Overlap between training examples and evaluation examples that can make apparent instruction-following gains unreliable.

Refusal data Demonstrations that teach the model to decline disallowed requests while preserving useful responses to safe neighboring requests.

Regression set A held-out evaluation suite used to detect capability loss after adaptation.

9.9 Exercises

1. Given a two-turn chat with a system message, two user turns, and two assistant turns, write a serialized template and mark every token span with $m_t = 0$ or $m_t = 1$ for Equation 9.2. State whether previous assistant turns are targets or context only.
2. Build a small audit table for 100 instruction examples. Include task type, language, source, target-token length, refusal category, and whether an automatic validator exists. What distributional gaps are visible?
3. Design a synthetic-data prompt for generating NL2SQL examples. Specify two automatic filters and one execution-based validator that would reject bad examples before training.
4. Reproduce the audit plan for a Self-Instruct-style run: list the seed tasks, similarity threshold, invalid-generation rules, and final instruction/instance counts you would report.
5. A model improves on an instruction-following leaderboard but gets worse at translation and arithmetic. List three plausible causes in the SFT mixture or training loop and one test for each cause.
6. Create three near-neighbor safety examples: one request that should be refused, one that should be answered, and one that requires clarification. Explain which surface features should not determine the policy decision.

7. Compare full fine-tuning and LoRA for the same SFT set. Report trainable parameters, checkpoint size, held-out instruction score, and one regression metric from the base model.

Chapter 10
Parameter-Efficient Adaptation

Abstract This chapter studies parameter-efficient adaptation as a tradeoff between expressivity, memory, storage, and operational control. It covers adapters, prefix tuning, prompt tuning, LoRA, QLoRA, target-module selection, rank choice, quantization, adapter composition, domain adaptation tradeoffs, and evaluation of adapted models against base-model regressions.

Chapter contract.

The reader should leave this chapter able to choose an adaptation method for a constrained deployment, justify rank and target-module choices, distinguish memory savings from latency or governance savings, and design evaluations that reveal when a small adapter has changed high-risk behavior.

10.1 What Is Being Made Efficient?

Full fine-tuning updates every parameter of a pretrained model. This is expressive, but it is expensive in three different ways. Training must store gradients and optimizer states for the full model; deployment may need a separate full checkpoint per task; and the adapted model may drift away from general capabilities learned during pretraining. Parameter-efficient fine-tuning, or PEFT, restricts the trainable variables while leaving most pretrained weights frozen.

Let θ_0 be the frozen base model and ϕ be the trainable adaptation parameters. PEFT solves

$$\begin{aligned} \mathrm{NLL}_{\mathcal{D}}(\theta_0, \phi) &= -\mathrm{avg}_{(x,y)\in\mathcal{D}} \log p_{\theta_0,\phi}(y \mid x), \\ \phi^\star &= \arg\min_{\phi} \mathrm{NLL}_{\mathcal{D}}(\theta_0, \phi), \qquad |\phi| \ll |\theta_0|. \end{aligned} \tag{10.1}$$

The inequality is the source of the savings, but it is also the source of the limitation. A PEFT method can only express changes reachable through the adapter subspace,

prompt vectors, scaling vectors, or low-rank matrices it introduces. A small adapter may be enough to teach a format or a narrow domain. It may be insufficient to repair a weak tokenizer, add broad factual knowledge, or reverse a harmful pretraining bias.

Efficiency therefore has to be specified. Training memory depends on which parameters require gradients and optimizer states. Checkpoint storage depends on whether the base model is shared and only adapters are saved. Serving latency depends on whether the adapter adds extra computation, extra sequence length, or unmerged matrix products. Operational complexity depends on how many adapters must be versioned, loaded, combined, audited, and rolled back.

10.2 Adapter and Prompt Families

The original adapter idea inserts small trainable modules inside each Transformer block while freezing the pretrained weights [59]. A common bottleneck adapter maps a hidden vector $h \in \mathbb{R}^d$ down to a smaller dimension b, applies a nonlinearity, maps back to d, and adds the result residually:

$$\text{Adapter}(h) = h + W_{\text{up}}\,\sigma(W_{\text{down}}h), \qquad W_{\text{down}} \in \mathbb{R}^{b\times d},\ W_{\text{up}} \in \mathbb{R}^{d\times b}. \tag{10.2}$$

When $b \ll d$, the adapter contributes far fewer parameters than the surrounding attention and feed-forward layers. The price is that every forward pass must execute extra projections unless the architecture provides a way to fold them into nearby operations, which is not generally true because of the nonlinearity.

Prefix tuning moves the trainable parameters from the residual stream into attention inputs. It learns continuous prefix vectors that act like task-specific key and value states available to the model at each layer [88]. The base model stays frozen, and generated tokens can attend to the learned prefix as if it were additional context. Prompt tuning is even simpler: it learns a small sequence of continuous input embeddings while leaving the model unchanged [83]. Prompt tuning is cheap to store and easy to swap, but it consumes effective context length and may be less expressive at smaller model scales.

LLaMA-Adapter is a useful hybrid between prompt methods and adapter methods. It keeps the LLaMA weights frozen, injects learnable adaptation prompts into higher transformer layers, and uses zero-initialized attention gates so undertrained prompts have little effect at the beginning of fine-tuning [198]. This design turns stability into an explicit mechanism: the model starts close to the frozen base and gradually admits instruction cues as the gates learn. A report for such methods should state which layers receive prompts, prompt length, gate initialization, trainable parameter count, and whether visual or other non-text tokens are added to the prompt path.

Adapter names are not enough to define the trainable state. LLaMA-Adapter v2-style implementations may also make RMSNorm parameters and per-linear scale or bias vectors trainable, in addition to prompt embeddings and gates. A checkpoint described as “adapter-only” should therefore list the saved parameter substrings or module classes, not just the method family name.

Table 10.1 Main families of parameter-efficient adaptation methods.

Method	Trainable Object	Main Deployment Cost
Adapters	Bottleneck modules inside blocks	Extra projections and adapter routing
Prefix tuning	Per-layer continuous key/value prefixes	More attention states and prefix management
Prompt tuning	Input-level soft prompt embeddings	Consumes context budget; task-specific prompt loading
IA^3-style scaling	Learned activation scaling vectors	Small extra elementwise operations
LoRA	Low-rank additive updates to selected matrices	Extra low-rank matmuls unless merged
QLoRA	LoRA on a quantized frozen base	Quantization kernels and possible merge constraints

Other PEFT methods use multiplicative gates or sparse updates. IA^3, for example, learns vectors that rescale internal activations and can be highly parameter-efficient in few-shot settings [98]. The common principle is that the base model supplies a rich representation space and the adapter supplies a low-dimensional control signal.

Table 10.1 compares the main PEFT families by trainable object and deployment cost.

10.3 LoRA

Low-Rank Adaptation, or LoRA, is the most widely used PEFT method for decoder large language models because it fits the linear algebra of Transformer blocks [61]. For a frozen weight matrix $W_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, LoRA learns a low-rank update

$$W' = W_0 + \Delta W, \qquad \Delta W = \frac{\alpha}{r} BA, \tag{10.3}$$

where $A \in \mathbb{R}^{r \times d_{\text{in}}}$, $B \in \mathbb{R}^{d_{\text{out}} \times r}$, rank r is small, and α is a scaling hyperparameter. The trainable parameter count for this matrix is

$$|\phi_W| = r(d_{\text{in}} + d_{\text{out}}), \tag{10.4}$$

instead of $d_{\text{in}} d_{\text{out}}$. For square matrices with width d, the fraction is approximately $2r/d$.

The initialization is part of the method. A typical implementation initializes one low-rank factor randomly and the other to zero so that $\Delta W = 0$ at the start of training. The initial model is exactly the base model, and adaptation begins as a learned residual update. LoRA dropout can be applied on the adapter path, and the base weight remains frozen.

Modern PEFT libraries make this contract more detailed than the phrase rank plus target modules. A reproducible `LoraConfig` should record initialization, scaling rule, saved non-adapter modules, layer/rank/alpha patterns, token-specific train-

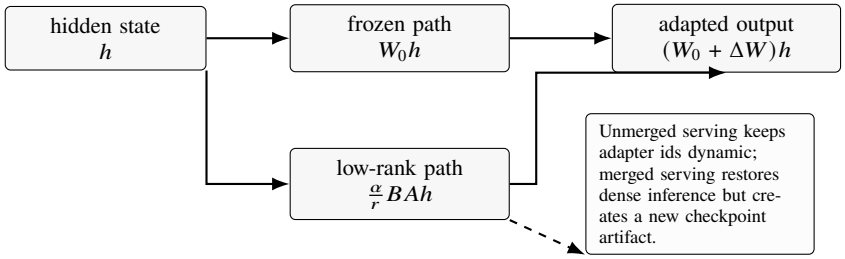

Fig. 10.1 LoRA adds a trainable low-rank residual to selected frozen matrices. The same learned update has different operational consequences depending on whether it is kept as an adapter or merged into the base weight.

ing, and whether the adapter targets modules or raw parameters [65]. Data-driven or decomposition-based initializers such as EVA, PiSSA, CorDA, OLoRA, and LoftQ change the starting point, while rsLoRA changes the scaling from α/r to $\alpha/\sqrt{r}$. The DoRA variant adds a learned magnitude path on top of the LoRA direction and can improve low-rank quality, but it also changes inference overhead and merge policy. Activated LoRA applies the adapter only after an invocation token sequence, which can reuse the base-model KV cache before that boundary but cannot be merged like an ordinary adapter. These options are useful, but they also expand the run card: one must state the invocation tokens, initialization dataset, mergeability, runtime-only configuration, and whether MoE expert tensors are reached through `target_parameters`.

The common SVD explanation is useful but limited. If a full fine-tuning update or a gradient snapshot has rapidly decaying singular values, then a low-rank adapter can capture many of the dominant update directions with far fewer parameters. That observation does not prove that every useful adaptation is low-rank, nor that the LoRA update learned under frozen weights matches the top singular vectors of a hypothetical full update. It is a modeling hypothesis that should be tested by varying rank, target modules, and data scale, then comparing task gains and regression losses.

Figure 10.1 shows how the frozen path and low-rank residual path combine at inference time.

LoRA can target different matrices. Attention projections are common: query, key, value, and output projections each control different aspects of attention behavior. Many instruction-tuning recipes target query and value projections only for lower cost; stronger adaptation often also targets key, output, and feed-forward projections. There is no universal best target set. Code tasks, multilingual adaptation, and structured generation may benefit from broader targets; a small style adapter may not.

Implementation layout can change what a target module means. In lightweight LLaMA code, query, key, and value may be stored as one fused `c_attn` projection. A Q/V-only LoRA adapter over that fused matrix must train low-rank factors only for the enabled slices, zero-pad the disabled key slice when forming ΔW, and apply the same mask during merge and unmerge. The run card should record the fused or split QKV layout, enabled slices, scaling, dropout, bias policy, and merge state; otherwise an adapter checkpoint can silently attach to the wrong columns even when tensor shapes load.

Embedding tables and the language-model head are also linear maps, so they can be adapted, but they deserve extra caution. Updating embeddings may help when new special tokens, domain vocabulary, or multilingual token usage must be learned. Updating the output head can change lexical probabilities directly. If input embeddings and the output head are tied, adapting one side can implicitly affect the other. A PEFT report should therefore state whether embeddings or the LM head were targeted, whether weights were tied, and whether the adapter was mergeable without changing tokenizer compatibility.

Chinese LLaMA-style adaptation is a useful counterexample to the idea that LoRA always means "freeze everything except two attention matrices." In that setting, LoRA handled broad attention and MLP updates, while the newly resized embeddings and LM head remained trainable so the added Chinese vocabulary could become useful [29]. The design separates three questions: which base weights are frozen, which low-rank paths are trainable, and which vocabulary-dependent dense matrices must still move. A report should answer all three, especially when a later instruction adapter is expected to sit on top of the language-adapted base.

Merging is a practical advantage. At inference time one can compute $W_{\text{merged}} = W_0 + (\alpha/r)BA$ and serve the model as an ordinary dense model, provided the precision and deployment policy allow the merge. This removes adapter matmul overhead. Keeping adapters unmerged allows dynamic swapping, multi-tenant serving, and combining adapters, but it requires runtime adapter management. Multiple adapters can be added as sums of low-rank updates only when their target matrices, scaling, and licensing constraints are compatible; empirical interference still has to be tested.

10.4 QLoRA and Quantized Training

LoRA reduces trainable parameters but does not by itself remove the memory needed to store the frozen base model. QLoRA attacks that cost by keeping the base model in a 4-bit quantized representation while backpropagating through it into LoRA adapters [35]. The base weights are not updated; gradients update only the low-rank matrices. Activations, adapter weights, and some computation may still use higher precision, but the dominant model-weight storage is much smaller.

A rough memory comparison for a model with N parameters is

$$M_{\text{full}} \approx M(W) + M(\nabla W) + M(\text{Adam}_1) + M(\text{Adam}_2), \tag{10.5}$$

where the optimizer states can exceed the parameter memory. In QLoRA, the dominant terms become

$$M_{\text{QLoRA}} \approx M_{\text{4bit}}(W_0) + M(\phi) + M(\nabla\phi) + M(\text{optimizer}(\phi)) + M(\text{activations}). \tag{10.6}$$

The last term can still be large, especially with long sequences and large batches, so gradient checkpointing and sequence packing remain important.

Quantization is not free. A 4-bit base model introduces quantization error, kernel constraints, and sometimes slower wall-clock training if hardware kernels are inefficient. QLoRA uses NormalFloat 4-bit quantization, double quantization, and paged optimizers to make the recipe practical, but those techniques do not guarantee that every model and task will match full fine-tuning. Evaluation must compare task quality, regression behavior, and calibration, not just whether the run fits on one GPU.

These three pieces should be reported separately. NF4 is a storage format chosen for approximately normal pretrained weights; it does not mean all computation happens in four bits. A typical QLoRA path stores frozen blocks in NF4, dequantizes to a higher computation dtype for matrix multiplies, and sends gradients through the dequantized path into LoRA matrices. Double quantization reduces scale and zero-point overhead by quantizing the per-block quantization constants; its benefit depends on block size and scale storage. Paged optimizers target peak-memory spikes rather than average memory by letting optimizer states move through unified memory when long sequences or checkpointing make activations peak. A reproducible QLoRA report should state storage dtype, compute dtype, block size, double-quant setting, target modules, whether adapters cover all linear layers or only attention projections, gradient checkpointing, paged optimizer, maximum sequence length, and peak memory.

NF4 should also not be treated as a universal theorem about four-bit codes. In absmax blockwise quantization, the distribution after dividing by the block maximum depends on the block size, so a fixed set of codepoints cannot be optimal for every block size [191]. This does not invalidate QLoRA's empirical recipe, but it makes the quantization report more important: an implementation should name the exact quantization type, block or group size, scale storage, and dequantization path. In common BitsAndBytes-style QLoRA setups, the concrete knobs include `load_in_4bit`, `bnb_4bit_quant_type`, `bnb_4bit_compute_dtype`, `bnb_4bit_use_double_quant`, and whether a paged optimizer such as `paged_adamw_32bit` is used.

Low-bit loading is also a placement contract. A script that sets `load_in_8bit=True`, `device_map=auto`, and a per-GPU `max_memory` cap is not only changing arithmetic precision; it is deciding where layers live, how much headroom is reserved, and whether CPU offload or uneven device placement can affect latency. Likewise, QLoRA training scripts often combine `bits=4`, NF4, double quantization, BF16 compute, gradient checkpointing, grouped-by-length batching, and a paged optimizer. Reporting only "4-bit LoRA" loses the actual systems recipe.

The current PEFT quantization guide makes LoftQ and target selection concrete [66]. Before training, the quantized model should be prepared for k-bit training, and the report should preserve the exact `BitsAndBytesConfig`: `load_in_4bit`, `bnb_4bit_quant_type`, double quantization, and compute dtype. For LoftQ to affect the quantization error broadly, LoRA generally needs to target as many linear layers as possible, often with an all-linear target-module setting rather than only query and value projections. The convenient `replace_lora_weights_loftq` path updates only the LoRA weights in one step, keeps the original quantized base weights, requires safetensors, and currently targets bitsandbytes 4-bit models. That tradeoff is attractive for compatibility, but it is not the same experiment as jointly iterating LoRA and quantized base weights.

10.5 Choosing Rank, Targets, and Hyperparameters

Rank r controls capacity. A rank that is too small underfits; a rank that is too large consumes more memory and may overfit a small dataset. The scaling parameter α controls the magnitude of the update relative to the base matrix. Practitioners often hold α/r or α in a conventional range, but the stable choice depends on optimizer, target modules, dataset size, and whether the base is quantized.

Target selection is usually more important than small changes in rank. If only the value projection is adapted, the model can alter what information is passed forward but has less control over how attention queries are formed. If feed-forward matrices are adapted, the adapter can change tokenwise transformations and often gains expressive power at a higher parameter cost. A defensible experiment grid varies one factor at a time: rank, target modules, learning rate, dropout, and number of training tokens.

Learning rates for adapters are often higher than full fine-tuning rates because the trainable matrices are newly initialized and small. This does not mean larger is always better. Too high a learning rate can produce brittle format imitation, loss spikes, or safety drift. A useful training log reports trainable parameter count, peak memory, tokens per second, gradient norm, training loss, validation loss, and task metrics.

The data collator remains critical. PEFT does not fix incorrect label masks, broken padding, or chat-template mismatches. A LoRA training implementation that freezes the base model, enables gradient checkpointing, disables the inference cache during training, pads to the longest sequence in the batch, and saves only trainable adapter weights captures the right systems shape. The textbook lesson is broader than the implementation: adapter training still inherits every preprocessing and evaluation obligation from SFT.

Concrete QLoRA SFT scripts make this contract visible. Source and target lengths are often truncated separately, for example with distinct source and response token budgets; a `train_on_source` flag decides whether prompt tokens receive labels or are filled with an ignore index. Grouping examples by length changes memory use and sometimes throughput, so it belongs in the run card. Adapter-only checkpoints are another trap: if resumption reloads adapter weights but not optimizer and scheduler state, the resumed curve is not the same experiment. A report should say whether optimizer state, scheduler state, random state, tokenizer changes, and adapter weights all survive save-and-resume.

10.6 Systems Costs and Deployment

PEFT changes the deployment architecture. With full fine-tuning, each adapted model is a complete checkpoint. With LoRA or adapters, many tasks may share one base checkpoint plus small task-specific files. This is attractive for storage and governance: a base model can be approved once, while adapters are reviewed and versioned separately. It also creates runtime challenges.

If adapters are loaded dynamically, the server must batch requests that use different adapter ids. A batch containing many adapters may lose some efficiency because each request needs a different low-rank path or merged weight set. If adapters are premerged, the server regains dense inference speed but loses cheap swapping and may need many resident model copies. If the base model is quantized, merging can require dequantization or a specialized quantized-merge path.

Adapters also appear inside training systems. In a memory-constrained RLHF loop, one frozen base can support a trainable policy adapter, a frozen reward-model adapter, and a reference path for KL scoring. The benefit is that the system avoids loading several full model copies. The risk is role confusion: reward scoring, KL computation, and policy updates are all ordinary forward passes unless the runtime records which adapter is active. Adapter id, score head, train/eval mode, frozen parameters, and merge state therefore belong in the checkpoint and training log.

Adapter composition is another operational risk. Two adapters trained separately may both improve their own tasks but degrade each other when summed or applied sequentially. A domain adapter that changes style can interfere with a safety adapter that depends on precise refusal wording. A multilingual adapter can improve target-language fluency while degrading English regression tests. Composition should be treated as a new model release, not as a bookkeeping operation.

LoRA merging has the same release risk. Platypus-style model building fine-tuned LoRA modules on a curated STEM/logical-reasoning mixture and then evaluated merges with other fine-tuned models, while explicitly checking training data for benchmark leakage [79]. The practical rule is that a merged model inherits both adapters' data and evaluation obligations. Similar leaderboard scores do not imply that two adapters will compose well; the merge must be tested by domain slice, safety slice, contamination audit, and regression benchmark.

Finally, PEFT does not remove the need for data security. An adapter can memorize sensitive examples even if it is small. Shipping only adapter weights is not a privacy guarantee. Membership inference, prompt extraction, and direct memorization tests are still relevant when the fine-tuning set contains private documents or user conversations.

10.7 Evaluation of Adapted Models

A PEFT report should include four classes of metrics. First, task metrics measure the reason for adaptation: exact match, execution accuracy, pass rate, retrieval-grounded answer quality, or human preference. Second, regression metrics measure what should not have changed: general chat quality, multilingual behavior, math, code, safety, and calibration. Third, systems metrics measure whether PEFT achieved the intended efficiency: trainable parameters, peak memory, training time, checkpoint size, tokens per second, and serving latency. Fourth, robustness metrics test sensitivity to prompt paraphrases, longer contexts, unseen schemas, and adversarial examples.

Comparisons must be fair. A LoRA model trained for 10,000 target tokens should not be compared to a full fine-tune trained for 10 million target tokens without accounting for data. A QLoRA run should not be declared equivalent to full fine-tuning because

Table 10.2 Example LoRA parameter counts for one square projection with width $d = 4096$.

Rank r	LoRA parameters $2rd$	Full matrix parameters d^2	Fraction
4	32,768	16,777,216	0.20%
8	65,536	16,777,216	0.39%
16	131,072	16,777,216	0.78%
64	524,288	16,777,216	3.12%

one benchmark matches. A prompt-tuned model should not be penalized for consuming context length unless the deployment actually has a tight context budget. The question is not which PEFT method is globally best; it is which method gives enough adaptation under the memory, latency, storage, and governance constraints of the application.

QLoRA's large experiment grid also shows a PEFT-specific trap: cheap training makes it easy to run many datasets, but dataset suitability can dominate dataset size. A small high-quality instruction set may beat a much larger mismatched mixture. PEFT reports should therefore state not only number of examples but source, filtering, task match, language and domain coverage, deduplication, and whether benchmark prompts or style leaked into tuning.

Table 10.2 gives a concrete scale check for common ranks in one square projection.

10.8 Key Terms

Adapter A small trainable module inserted into a frozen pretrained network, often using a bottleneck projection and residual connection.

Zero-gated prompt adapter A prompt-style adapter whose contribution is initially gated near zero so the frozen base path dominates early training.

LoRA A low-rank additive update to selected weight matrices, usually mergeable into the base weights for inference.

rsLoRA A LoRA scaling variant that uses $\alpha/\sqrt{r}$ rather than α/r to make higher-rank adapters easier to train.

DoRA A weight-decomposed LoRA variant that learns magnitude separately from the low-rank direction and therefore changes both quality and inference overhead.

Activated LoRA A LoRA variant that activates only after an invocation token sequence, enabling base-cache reuse before the boundary but preventing ordinary merging.

QLoRA A recipe that trains LoRA adapters while storing the frozen base model in a low-bit quantized format.

LoftQ A LoRA initialization strategy for quantized training that initializes adapter weights to reduce quantization error.

NF4 A 4-bit storage format used by QLoRA for approximately normal pretrained weights before dequantized computation; its behavior still depends on block size and scale handling.

Double quantization Quantization of quantization constants, reducing the metadata overhead of low-bit block quantization.

Paged optimizer An optimizer implementation that uses unified-memory paging to absorb activation-driven peak-memory spikes.

Rank The dimension r of the low-rank update in LoRA, controlling adapter capacity and parameter count.

Target modules The model matrices or layers to which adapters are applied, such as attention projections or feed-forward projections.

Target parameters Raw parameter tensors, such as some MoE expert tensors, that receive LoRA updates even when there is no `nn.Linear` module to wrap.

Trainable token indices A PEFT configuration that tunes selected embedding rows without retraining the entire embedding matrix.

Merge The process of adding a trained LoRA update into the base weight matrix for ordinary dense inference.

10.9 Exercises

1. For a square projection matrix with width 4096, compute the LoRA parameter count for ranks 4, 8, 16, and 64. Compare each to the full matrix parameter count.
2. Implement Equation 10.3 for one linear layer. Initialize the adapter so that the adapted layer is initially identical to the base layer, then verify the equality numerically.
3. Fine-tune the same small model with LoRA on query/value projections only and on all attention projections. Report trainable parameters, validation loss, task metric, and one regression metric.
4. Design a QLoRA memory budget for a 7B-parameter model. List which terms come from quantized base weights, adapters, optimizer states, gradients, and activations.
5. Write the reproducibility checklist for a QLoRA run, including storage dtype, compute dtype, block size, double quantization, target modules, gradient checkpointing, paged optimizer, maximum sequence length, peak memory, and dataset filtering.
6. Design an evaluation plan for merging two LoRA adapters. Include domain-slice gains, regression tasks, safety checks, and benchmark-contamination checks for both source datasets.
7. Choose a domain task where prompt tuning might be preferable to LoRA and one where LoRA might be preferable to prompt tuning. Justify the choice using deployment constraints.
8. Suppose two adapters improve different tasks separately but fail when combined. Propose three diagnostic experiments to distinguish rank interference, conflicting data, and prompt-template mismatch.
9. Write a modern LoRA configuration manifest for a QLoRA run. Include initialization, rsLoRA or DoRA choices, LoftQ path, module targets versus raw-parameter targets, trainable token indices, mergeability, quantization config, and runtime-only options.

Chapter 11
Domain and Language Adaptation

Abstract This chapter studies how LLMs are adapted to Chinese, domain-specific corpora, NL2SQL, medical-style retrieval tasks, and other specialized settings. It compares continual pretraining, supervised fine-tuning, retrieval, tokenizer extension, adapter-based updates, evaluation under domain shift, and governance boundaries for private or high-stakes knowledge.

Chapter contract.

The reader should leave this chapter able to diagnose whether a domain or language gap is caused by data, vocabulary, retrieval, task format, or evaluation shift, and then choose a lower-risk adaptation path before changing model weights.

11.1 Adaptation Is a Diagnosis

Domain adaptation is not one method. It is a diagnosis of what the base model lacks. A model may fail in a new setting because the tokenizer fragments the language badly, because the relevant facts were absent or stale during pretraining, because the domain uses unfamiliar style and discourse conventions, because the task requires a structured output grammar, or because the answer must be grounded in private documents that should never be memorized into model weights. Each diagnosis points to a different intervention.

Table 11.1 gives a practical decision map. Continual pretraining, SFT, LoRA, tokenizer extension, and retrieval are complementary tools, not substitutes with the same risk profile. The cheapest intervention that addresses the actual failure should be tried before a more invasive one.

The most common error is to call every failure a fine-tuning problem. If the user asks about a private handbook updated last week, fine-tuning is the wrong default: retrieval gives freshness and access control. If the model cannot produce valid SQL for a known

Table 11.1 Decision map for diagnosing domain adaptation needs.

Observed Failure	Likely Cause	Candidate Intervention
Many tokens per word or character	Tokenizer mismatch	Vocabulary extension or a language-specialized tokenizer
Knows style but not facts	Missing or changing knowledge	Retrieval first; continual pretraining only for stable public corpora
Correct facts, wrong format	Interface or output contract missing	SFT or PEFT on demonstrations with validators
Fails on schemas or APIs	Poor grounding in structured context	Schema-aware prompting, SFT, execution feedback, or tool use
Good benchmark, bad deployment	Distribution shift or contamination	Private held-out evaluation and slice analysis
Unsafe advice in high-stakes domain	Policy and evidence boundaries unclear	Domain safety data, abstention rules, expert review, and monitoring

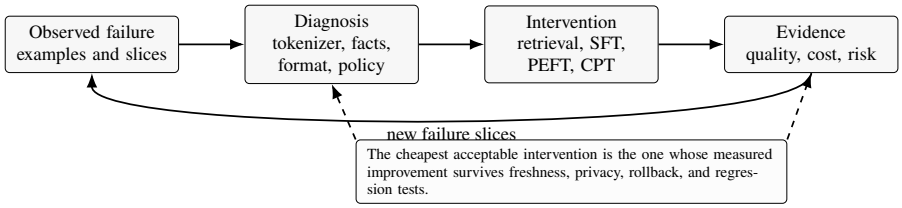

Fig. 11.1 Domain adaptation is a diagnosis loop. The same symptom can call for retrieval, supervised adaptation, continued pretraining, tokenizer work, or a governance change.

schema, supervised examples and execution validators may be appropriate. If the model uses too many tokens for a target language, tokenizer and continued pretraining choices matter before instruction style. If the application is medical, legal, or financial, the adaptation plan must include evidence boundaries and expert evaluation, not only a higher score.

Figure 11.1 turns that rule into a diagnosis loop that can be repeated as new failure slices appear.

11.2 Language Adaptation

Language adaptation starts with tokenization. A subword tokenizer trained mostly on English may split Chinese, Thai, Arabic, or domain-specific symbols into long sequences. This increases context cost and makes long-range modeling harder. A simple diagnostic is token fertility:

$$F_L = \mathrm{avg}_{s \in \mathcal{S}_L} \frac{\mathrm{tokens}(s)}{\mathrm{units}(s)}, \tag{11.1}$$

where units(s) may be characters, words, or language-appropriate segments. Fertility should be compared across languages and domains using representative text. High fertility does not prove poor model quality, but it is an early warning for cost and coverage.

There are three broad tokenizer strategies. The first is to keep the original tokenizer and adapt only weights. This is simple and preserves checkpoint compatibility, but it may waste context. The second is to extend the vocabulary with frequent target-language or domain tokens. New embeddings can be initialized from the average or composition of their old subpieces, then trained during continued pretraining. The third is to train a new tokenizer and convert or retrain the model. This can improve efficiency but is the most disruptive because embeddings, output heads, and compatibility with existing checkpoints change.

Vocabulary extension has an implementation contract. The tokenizer should be trained on representative target-language text, commonly with a subword method such as SentencePiece [76]; the new vocabulary should preserve old token ids when checkpoint compatibility matters; and the embedding matrix and language-model head must be resized from $V \times H$ to $V' \times H$. If input and output embeddings are tied, both sides of the tie must remain consistent. The newly appended rows should be trained with continued pretraining before instruction tuning, preferably with replay data so the model does not forget the original language distribution. Chinese LLaMA and Alpaca followed this pattern by adding 20,000 Chinese tokens to LLaMA, producing a 49,953-token merged vocabulary and substantially reducing Chinese sequence length before Chinese continued pretraining and instruction tuning [29].

The Chinese LLaMA recipe also shows what a vocabulary-extension run card should contain. The new Chinese tokenizer was merged by vocabulary union, so old LLaMA token ids remained stable and new rows were appended to the embedding and LM-head matrices. The 7B basic run used a staged adaptation in which the newly added embeddings were trained before broader LoRA-based updates; later runs used LoRA over attention and MLP paths while keeping embeddings and the LM head trainable. Instruction-tuned Chinese Alpaca adds a padding token, so its serving stack must know that the tokenizer vocabulary differs from the base Chinese LLaMA tokenizer. These details matter for adapter reuse, checkpoint merging, perplexity comparison, and any regression test that mixes old English prompts with new Chinese prompts.

Chinese adaptation illustrates the tradeoff. Chinese LLaMA and Alpaca-style work added Chinese text encoding, secondary pretraining, and Chinese instruction tuning to make LLaMA-family models more useful for Chinese users [29]. The lesson is not that every language needs a separate model. The lesson is that language coverage is a full pipeline: tokenizer fertility, corpus selection, continued pretraining, instruction data, safety examples, and evaluation all matter.

Baichuan 2 shows the train-from-scratch version of the same lesson. Instead of extending an English-heavy checkpoint, it trains multilingual 7B and 13B models on a 2.6-trillion-token mixture and designs the tokenizer for Chinese, English, code, academic text, and numeric data from the beginning [184]. This can produce stronger language and domain coverage, but it is much more expensive than adaptation. A release claim should therefore distinguish tokenizer extension, continued pretraining,

and full multilingual pretraining: they solve related problems at very different cost, compatibility, and governance levels.

Multilingual evaluation must be slice-based. A single averaged score can hide a model that improves simplified Chinese but degrades English, improves news text but fails colloquial dialogue, or answers in the wrong language. Evaluation should include target-language perplexity, instruction following, translation or cross-lingual transfer where relevant, code-switching, culturally specific references, and safety examples written by native speakers.

11.3 Continual Pretraining

Continual pretraining, also called domain-adaptive pretraining when the corpus is domain-specific, continues the language-model objective on new unlabeled text. It is useful when the domain has stable terminology, discourse patterns, and public factual regularities. "Do not stop pretraining" showed that continuing pretraining on domain and task-related text can improve downstream performance for domain tasks [52]. The same idea scales to modern decoder models, but the risks scale too.

A common objective mixes domain data with replay data from the general distribution:

$$\mathcal{L}_{\mathrm{CPT}}(\theta) = \lambda \mathbb{E}_{x \sim \mathcal{D}_{\mathrm{domain}}}[-\log p_\theta(x)] + (1-\lambda)\mathbb{E}_{x \sim \mathcal{D}_{\mathrm{replay}}}[-\log p_\theta(x)]. \tag{11.2}$$

The replay term reduces forgetting. Without it, a narrow corpus can make the model better at local terminology while degrading general language quality, multilingual coverage, or safety behavior. The mixture coefficient λ, learning rate, number of tokens, and deduplication policy are central hyperparameters.

Continual pretraining should not be used to inject private or rapidly changing facts unless there is a strong reason to memorize them into weights. Weight updates are hard to audit, hard to revoke, and hard to access-control. Retrieval is usually better for internal policies, customer records, current prices, medical guidelines that update, or legal documents that differ by jurisdiction. Continual pretraining is better suited to stable public corpora such as scientific literature, code conventions, patent language, or target-language web text whose licensing has been reviewed.

Biomedical models illustrate both value and caution. BioGPT used large-scale biomedical literature to train a generative model specialized for biomedical text generation and mining [104]. Such specialization can improve terminology and task performance, but it does not by itself make the model clinically safe. Biomedical language competence is different from validated medical decision support.

Medical fine-tuning demos make the same point in pipeline form. One stage may use encyclopedia and textbook text for continued pretraining, another may use doctor-patient dialogues for SFT, and a reward-model stage may compare a human doctor answer against a model-generated answer. These are not interchangeable data sources. The report should state which source feeds each stage, whether examples were split by document, patient case, time, or only by random row, and whether adjacent chunks from

the same textbook or dialogue can appear on both sides of the split. A random 5 percent validation split is useful for debugging loss, but it is weak evidence for deployment generalization in medicine.

11.4 Supervised Domain Tuning

SFT or PEFT is appropriate when the missing capability is a task interface rather than broad language modeling. A customer-support assistant may need to follow a ticket triage format. A writing assistant for a fictional world may need to continue in a specific voice. An enterprise assistant may need to output a strict JSON object. A database assistant may need to map natural-language questions to SQL. These are supervised behaviors.

The training examples should separate three things: domain context, user request, and expected response. A useful schema for a writing task might contain a short memory summary, recent text, and the next passage. A useful schema for QA might contain question, evidence, and answer. A useful schema for NL2SQL contains the natural-language question, the database schema, optional foreign-key information, and the target SQL. Collapsing all fields into a raw string may work for a demo, but it hides the contract from preprocessing, validation, and evaluation.

Field semantics deserve their own audit. Preference datasets often contain names such as `response_chosen` and `response_rejected`; SFT datasets often contain only a target answer. Accidentally using the rejected response as the SFT demonstration trains the model toward behavior that the later reward model is supposed to penalize. A domain tuning run should therefore include a small rendered-sample check for every stage: pretraining text, SFT prompt-answer string, reward-model chosen/rejected pair, and PPO query template.

PEFT is often the first adaptation method for supervised domain tuning because it is cheap to reverse and compare. LoRA can adapt a base model to a schema format or domain style while storing only adapter weights [61]. QLoRA can make the same experiment feasible under tighter hardware limits [35]. The danger is false confidence: a small adapter can overfit a small domain set and appear fluent while failing on unseen entities, new schemas, or adversarial wording.

Data splits must represent deployment. Randomly splitting examples in a domain dataset can leak entities, templates, or schemas. For an internal support assistant, split by customer or time. For a code assistant, split by repository. For fiction continuation, split by work or chapter. For NL2SQL, split by database when the deployment will see new schemas. The evaluation split should test the generalization claim the model is supposed to satisfy.

11.5 Structured Tasks: NL2SQL

Natural-language-to-SQL tasks expose the difference between fluent text and executable meaning. Given a question q and schema S, the model emits a SQL query y:

$$y \sim p_\theta(\cdot \mid \tau(q, S)). \tag{11.3}$$

The answer is correct only if the query is syntactically valid, refers to the right tables and columns, respects SQL dialect, and returns the intended result. Spider made cross-domain text-to-SQL evaluation important by splitting across databases and requiring generalization to unseen schemas [192].

Schema serialization is part of the model interface. A prompt that lists only table names is different from one that includes columns, types, primary keys, foreign keys, and example rows. Normalizing table names can help a model learn structural patterns instead of memorizing arbitrary identifiers, but it can also hide entity-linking difficulty. The preprocessing choice should match the intended evaluation. If deployment requires real table names, the model or postprocessor must recover them reliably.

Evaluation should include exact string match only as a diagnostic. SQL has many equivalent surface forms. Execution accuracy, where the predicted query is run against a database and compared to the expected result, is often more meaningful. It is still imperfect: two wrong queries can return the same result on a small database, and unsafe queries should not be executed outside a sandbox. A robust NL2SQL evaluation includes parse validity, execution accuracy, schema-linking accuracy, latency, and manual review of a sample of failures.

For production, an NL2SQL model is usually part of a guarded system. The system can restrict SQL to read-only statements, require a query planner or parser check, ask for clarification when the question underspecifies a join, and show the generated query before execution. Fine-tuning the model is only one layer of safety.

11.6 Retrieval Versus Weight Updates

Retrieval-augmented generation and fine-tuning solve different problems. Retrieval supplies evidence at inference time; fine-tuning changes behavior stored in parameters. If the knowledge is fresh, private, user-specific, or legally sensitive, retrieval is usually the safer first step. If the behavior is a stable output pattern or the domain language is broadly missing, fine-tuning may be appropriate. If both are missing, combine them: retrieve evidence and fine-tune the model to use evidence faithfully.

A medical-style QA system is a useful example. The knowledge base can store prior cases, guidelines, or approved documents; a retriever finds relevant evidence; a reranker improves precision; and the generator drafts a response conditioned on the selected evidence. This architecture can update the knowledge base without retraining the model, and it can log which evidence was shown. It also introduces new failure modes: the retriever can miss relevant passages, the reranker can prefer superficially similar but

clinically different cases, and the generator can overstate conclusions. Chapter 12 studies these retrieval failures in detail.

Fine-tuning on retrieved answers can improve style and evidence use, but it should not train the model to invent citations. The target answer should be traceable to evidence, and examples without adequate evidence should teach abstention or clarification. In high-stakes settings, a fluent answer without evidence is a failure even if it sounds domain-specific.

11.7 Safety and Governance in Domains

Domain adaptation often moves a model closer to consequential decisions. Medical, legal, financial, educational, and enterprise systems require stricter governance than general chat. The first question is scope: what is the model allowed to do, and what must remain with a qualified human or an external system? The second question is evidence: what sources can the model rely on, and how are they updated? The third question is accountability: what logs, audits, and rollback paths exist when the system fails?

Safety data should be domain-specific. A medical assistant needs examples that distinguish general health education from diagnosis, triage, emergency advice, dosage calculation, and clinician-facing summarization. A legal assistant needs jurisdiction, uncertainty, and non-attorney boundaries. A financial assistant needs suitability constraints and disclosure. Generic refusal data is not enough because domain risk often depends on subtle context.

Privacy is also central. A domain corpus may contain personal data, trade secrets, unreleased product plans, or copyrighted material. Deduplication and de-identification reduce risk but do not eliminate it. If private facts need to be revocable, retrieval with access controls is preferable to weight updates. If adaptation uses private examples, membership-inference and memorization checks should be part of release evaluation.

11.8 Evaluation Under Domain Shift

Domain adaptation should be evaluated by slices, not only by averages. A good report includes in-domain and out-of-domain metrics, pre-adaptation and post-adaptation comparisons, and failure analysis by entity type, language, length, source, and date. For language adaptation, report token fertility and target-language task quality. For structured tasks, report validity and execution. For retrieval-grounded tasks, report retrieval recall separately from answer quality. For high-stakes tasks, include expert review and abstention quality.

Contamination is a special concern. Domain benchmarks are often small and public. A model can memorize task templates or examples during continued pretraining or synthetic-data generation, especially when the same benchmark is used repeatedly to tune prompts and adapters. Evaluation sets should be held out before adaptation begins,

and their overlap with training data should be checked by exact match, near-duplicate search, and task-specific leakage tests [180].

The final question is operational: does adaptation improve the deployed workflow? A model that raises an NL2SQL benchmark by two points but produces unsafe queries is worse. A model that sounds more medical but cites no evidence is worse. A model that improves Chinese chat but doubles context cost because of tokenization may be uneconomical. Adaptation is successful only when model quality, systems cost, safety, and governance improve together.

Stage-by-stage smoke tests are part of this evaluation. If a continued-pretraining checkpoint answers a simple medical prompt with multilingual gibberish, repeated fragments, or broken special tokens, the next SFT or RLHF stage may hide the failure rather than fix it. Such outputs usually point to tokenizer mismatch, too little or too narrow training, an adapter/base merge problem, or an unstable generation configuration. The correct response is to debug the stage boundary before interpreting later reward or chat examples.

This can be written as a release criterion rather than a slogan. Let ΔQ be the measured task-quality gain, ΔC the added cost, ΔR the added risk estimate, and ΔG the governance burden created by data, licensing, monitoring, or rollback. A domain release should be justified only when

$$U_{\text{adapt}} = \Delta Q - \lambda_C \Delta C - \lambda_R \Delta R - \lambda_G \Delta G > 0$$

under weights chosen before the experiment. The equation is deliberately simple: it forces the team to admit that a benchmark gain is not enough when the adapted system is slower, harder to audit, or more dangerous.

11.9 Key Terms

Continual pretraining Continuing the language-model objective from a pretrained checkpoint on additional corpora.

Domain-adaptive pretraining Continual pretraining focused on a target domain such as biomedical papers, legal documents, or code.

Token fertility The number of tokenizer tokens needed per language or domain unit, used as a diagnostic for tokenizer coverage.

Vocabulary extension Adding target-language or domain tokens while resizing embeddings and output heads so a pretrained checkpoint can encode the new units.

Schema linking The process of connecting words in a natural-language query to database tables, columns, and values.

Execution accuracy The fraction of generated programs or SQL queries that return the expected result when executed in a controlled environment.

Replay data General-distribution data mixed into continual pretraining to reduce forgetting.

11.10 Exercises

1. Compute token fertility for 100 English sentences and 100 target-language sentences under the same tokenizer. Interpret the result as a cost and modeling diagnostic.
2. Design a vocabulary-extension experiment for Chinese or another target language. Specify tokenizer training data, new-token initialization, replay data, and compatibility tests for old prompts and adapters.
3. Design a continual-pretraining mixture for a legal-domain model. Specify domain sources, replay sources, deduplication rules, and three regression tests.
4. Build an NL2SQL evaluation plan for a new database. Include schema split, prompt serialization, parse validity, execution accuracy, and sandboxing rules.
5. Compare retrieval-only, LoRA-only, and retrieval-plus-LoRA baselines on a small domain QA set. Report retrieval recall separately from answer correctness.
6. Write three domain safety examples for a medical assistant: an allowed education question, a question requiring urgent referral, and a request the assistant should refuse or redirect.
7. A model improves in-domain perplexity after continual pretraining but gets worse at general instruction following. Propose two data-level causes and two training-level causes.

Part IV
Alignment, Applications, and Evaluation

Chapter 12
Retrieval, Tools, and Agents

Abstract This chapter treats retrieval, tools, and agents as control systems around an LLM rather than prompt decorations. It covers retrieval-augmented generation, embeddings, vector databases, retrievers, rerankers, chunking, citation faithfulness, context engineering, typed memory, tool calls, ReAct-style reasoning-action loops, permissions, and workflow evaluation.

Chapter contract.

The reader should leave this chapter able to specify a RAG or tool-using system as a set of retrieval, context, action, and rollback contracts, evaluate evidence use rather than answer fluency alone, and identify prompt-injection and memory risks introduced by external state.

12.1 RAG as a Control System

Retrieval-augmented generation, or RAG, is often described as putting documents into the prompt. That description is too small. A production RAG system is a control system around a large language model. It decides what corpus is searchable, how documents are chunked, how queries are rewritten, which retriever and reranker are trusted, how much context is inserted, how the answer should cite evidence, when the model should abstain, and how to defend the system against hostile text inside retrieved documents.

The basic decomposition is

$$q \to \text{retrieve}(q) \to E_k \to \text{generate}(q, E_k) \to a, \tag{12.1}$$

where q is the user query, E_k is a selected evidence set, and a is the answer. RAG originally connected neural retrievers with sequence generators for knowledge-intensive NLP tasks [85]. Dense passage retrieval then made learned embedding search a practical alternative to purely sparse lexical search for open-domain question answering [70].

Retrieval-augmented pretraining systems such as RETRO showed that retrieval can also be integrated into model training and scaling, not only into prompting [13].

The original RAG formulation is also useful as a probabilistic model. The retrieved passage z is a latent evidence variable with retriever distribution $p_\eta(z \mid x)$ and generator distribution $p_\theta(y_t \mid x, z, y_{<t})$. Let $z_1, \ldots, z_k$ denote the retrieved candidates and let $A_t(z) = p_\theta(y_t \mid x, z, y_{<t})$. RAG-Sequence assumes one retrieved document explains the whole answer:

$$\begin{aligned} p_{\text{seq}}(y \mid x) \approx\ & p_\eta(z_1 \mid x) A_1(z_1) \cdots A_T(z_1) \\ & + \cdots + p_\eta(z_k \mid x) A_1(z_k) \cdots A_T(z_k), \end{aligned}$$

whereas RAG-Token marginalizes evidence at each decoding step:

$$\begin{aligned} C_t &= p_\eta(z_1 \mid x) A_t(z_1) + \cdots + p_\eta(z_k \mid x) A_t(z_k), \\ p_{\text{tok}}(y \mid x) &\approx C_1 \cdots C_T. \end{aligned}$$

Modern production systems often materialize the selected evidence in a prompt instead of training this exact latent-variable model, but the distinction remains pedagogically important: evidence can be chosen once for the whole response, or it can vary across generated tokens. Multi-hop answers, synthesis across sources, and citation checking all depend on which assumption the system actually makes.

The original RAG system is also a concrete training recipe, not merely an inference pattern. It initializes the retriever from DPR, uses maximum inner product search over a dense passage index to obtain top-K passages, concatenates each passage with the input for a BART-style generator, and trains by minimizing the negative marginal log likelihood of the target sequence [85, 70]. No gold supporting document labels are required: the retriever is trained through the marginal likelihood signal. That convenience has a systems caveat. If document embeddings are updated during training, the vector index must be refreshed; if the document encoder and index are frozen, training is simpler but retrieval quality is bounded by the fixed memory. A report should therefore state which encoders are trainable, how often the index is rebuilt, what K is used, and whether decoding marginalizes documents during generation or only inserts top-ranked context into a prompt.

The central benefit is separation of concerns. Model weights store general linguistic and reasoning behavior; the retrieval layer stores mutable or private evidence. This separation improves freshness, auditability, and access control. It does not guarantee truth. If retrieval misses the relevant document, the generator may hallucinate. If retrieval returns misleading evidence, the generator may faithfully summarize the wrong source. If retrieved text contains instructions to ignore the user or reveal secrets, the generator may follow the wrong authority. The system must therefore evaluate retrieval, generation, and control logic separately.

Figure 12.1 lays out the controlled evidence path used for the rest of the chapter.

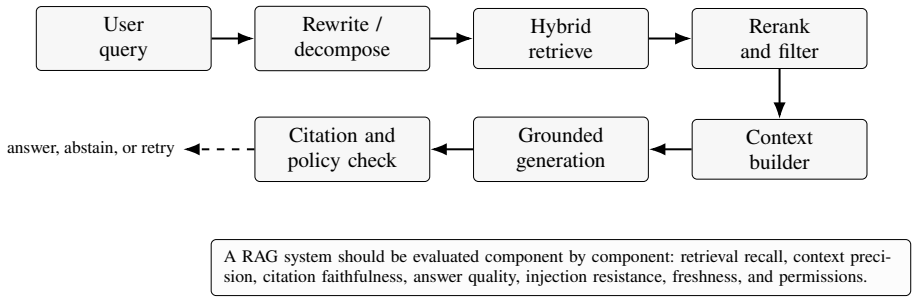

Fig. 12.1 A RAG pipeline as a controlled evidence path. The figure is an original synthesis from dense retrieval, RAG, and context-engineering literature [70, 85, 106].

12.2 Indexing and Retrieval

An index begins with documents, but retrieval usually operates over chunks. Chunking is a modeling decision. Short chunks improve lexical precision and citation granularity, but they can lose context. Long chunks preserve context, but they consume prompt budget and can bury the relevant sentence. Overlap helps when answers cross boundaries, but it increases storage and can produce duplicate evidence. A practical index stores each chunk with stable document id, source, timestamp, permissions, title, section path, and character offsets so that answers can cite and audit the original source.

Dense retrieval embeds queries and chunks into vectors. A typical scoring function is inner product or cosine similarity:

$$s_{\text{dense}}(q, d) = \langle f_q(q), f_d(d) \rangle, \qquad D_k(q) = \text{topk}_{d \in C}\, s_{\text{dense}}(q, d). \tag{12.2}$$

Sparse retrieval, such as BM25-style lexical search, remains valuable because exact names, rare identifiers, error codes, and legal citations often matter. Hybrid retrieval combines dense and sparse signals:

$$s_{\text{hybrid}}(q, d) = \lambda n_{\text{dense}}(q, d) + (1 - \lambda) n_{\text{sparse}}(q, d), \tag{12.3}$$

where scores are normalized before combination. The weight λ is a tuning parameter, not a constant of nature.

Query rewriting can improve recall. The system may expand acronyms, generate paraphrases, add domain synonyms, or split a multi-part question into subqueries. In a medical-style retrieval system, for example, a symptom description may be rewritten into several clinically adjacent phrasings before vector search. Rewriting also adds risk: a rewrite can change the user's intent. The original query and all rewrites should be logged, and evaluation should measure whether rewriting improves recall without increasing false positives.

Table 12.1 Design choices in a retrieval-augmented generation pipeline.

Component	Choice	Failure Mode
Chunking	Size, overlap, metadata, boundaries	Relevant evidence split or buried
Embedding	Model, normalization, language coverage	Poor recall for domain terms or languages
Retriever	Dense, sparse, or hybrid search	Misses exact ids or semantic paraphrases
Reranker	Cross-encoder, LLM judge, or rules	High precision but high latency or bias
Context builder	Ordering, deduplication, compression	Prompt budget wasted or evidence distorted
Generator	Prompt, citation rules, abstention	Unsupported claims or citation laundering
Monitor	Logs, freshness, drift metrics	Silent index decay or stale answers

12.3 Reranking and Context Construction

Retrievers are usually optimized for recall at a moderate k. The generator, however, needs a small set of highly relevant, non-duplicative chunks inside a finite context window. Reranking bridges this gap by scoring query-document pairs with a more expensive model after initial retrieval. A cross-encoder reranker can compare the query and chunk jointly, which is often more precise than independent embeddings, but it is slower and harder to run over a large corpus.

Context construction is not the same as taking the top five chunks. The system should remove near duplicates, respect source permissions, include enough neighboring context for interpretation, and order evidence deliberately. Long-context models do not always use middle-position evidence robustly; performance can be highest when relevant information appears near the beginning or end of the prompt [99]. Therefore, rank order, section grouping, and explicit citation markers can affect answer quality.

Table 12.1 lists design choices that should be documented in any serious RAG system.

12.4 Generation with Evidence

The generator should be instructed to answer from evidence, but instruction alone is insufficient. The prompt should separate system policy, user request, retrieved evidence, and tool outputs. Retrieved text is data, not authority. A document may contain malicious instructions, outdated statements, or user-provided content. The system prompt should state that retrieved documents are untrusted evidence and that higher-priority instructions come from the developer or system layer.

Grounded generation has three obligations. First, the answer should use the evidence that supports the claim. Second, the citation should point to the evidence actually used, not merely to a top-ranked chunk that seems related. Third, the model should abstain or ask for clarification when evidence is missing or conflicting. Recent RAG evaluation

surveys emphasize that answer quality, context relevance, citation faithfulness, and robustness should be measured separately [45].

Citation faithfulness is harder than appending source ids. A model can cite a source that does not support the sentence, cite a source that supports only part of the sentence, or synthesize across sources without making the connection clear. A useful answer format ties each factual claim or paragraph to evidence ids and permits "not found in the provided sources" as a valid outcome. For high-stakes domains, unsupported claims should be treated as failures even if the natural language is plausible.

Compression is another risk. Systems often summarize retrieved chunks to save context. If the compressor removes uncertainty, dates, exceptions, or source qualifiers, the generator receives distorted evidence. Compression should be evaluated as its own component, and the system should preserve links to the original text for audit.

12.5 RAG Evaluation

RAG evaluation begins before generation. Retrieval recall asks whether the evidence needed to answer the question appears in the candidate set. If recall is low, no generator can reliably fix the system. Metrics include recall@k, mean reciprocal rank, and normalized discounted cumulative gain when relevance labels are graded. For enterprise systems, labels may come from human annotation, click logs, answer citations, or synthetic questions generated from documents and then audited.

After context construction, evaluate context precision: how much of the inserted context is relevant? A prompt filled with weakly related chunks can lower answer quality and increase cost. Then evaluate answer quality: correctness, completeness, citation support, abstention behavior, and style. These metrics should be reported by slice: document source, document age, language, query type, entity frequency, and permission group.

End-to-end judge scores are useful for triage but dangerous as the only metric. A judge model can reward fluent hallucinations, miss citation errors, or share the same blind spots as the generator. Deterministic checks should be used where possible: exact source id, quote span overlap, executable answer, policy compliance, or database result. Human review remains necessary for subtle factuality and high-stakes claims.

The right baseline matters. Compare against no retrieval, sparse retrieval, dense retrieval, hybrid retrieval, reranking, and an oracle context where the gold evidence is supplied. The gap between oracle-context generation and retrieved-context generation estimates how much quality is being lost in retrieval and context construction. The gap between oracle-context generation and the gold answer estimates generator limitations.

12.6 Context Engineering and Memory

RAG is one form of a broader discipline: context engineering. Context engineering treats the entire information payload supplied to the model as an object of design:

system instructions, developer policy, user request, retrieved documents, conversation history, tool outputs, temporary scratchpads, long-term memory, and structured state. Recent surveys argue that this is no longer merely "prompt engineering"; it is the systematic construction, processing, compression, routing, and governance of context for model behavior [106].

The older phrase "prompt engineering" is still useful when it is interpreted operationally. In real applications it includes more than hand-writing a clever instruction: batch rewriting, deduplication, templating, role-play setup, chain-of-thought or scratchpad policy, data generation prompts, RAG context assembly, judge prompts, and task-specific output contracts. Treating these as code and data artifacts keeps them testable. A prompt change that rewrites user input, inserts retrieved context, or asks a judge to score an answer should be versioned and evaluated like any other model-facing transformation.

This distinction matters because a capable model can still fail when the context contract is wrong. A long prompt can contain the relevant evidence but place it where the model underuses it. A memory system can preserve a user's preference but also preserve outdated or sensitive information. A tool output can be accurate but stale by the time the model acts on it. A context compressor can reduce latency while deleting the uncertainty that should have caused abstention. The design unit is therefore not a prompt string; it is a stateful context pipeline.

Long-context models do not eliminate context engineering. Benchmarks such as ∞Bench, RULER, and LongBench v2 show that nominal context length differs from effective context use, especially when tasks require aggregation, multi-hop reasoning, contradiction handling, or realistic long-document understanding [199, 60, 10]. A model that accepts 256K or 1M tokens may still fail to bind evidence across sections, retrieve a non-literal fact, or ignore distractors. In many systems, a shorter prompt with better retrieval, chunking, and state management beats a raw long-context dump.

Memory should be typed. A production assistant may store user preferences, task state, conversation summaries, document embeddings, tool observations, and safety-relevant decisions. These should not share one undifferentiated text buffer. Each memory type needs provenance, update rules, expiration, user visibility, deletion semantics, and permission checks. A memory that cannot be inspected or revoked becomes a hidden training set at inference time.

The practical test is simple: if a later answer depends on a piece of context, the system should be able to say where that context came from, why it was allowed into the prompt, whether it was transformed, and how it can be removed or corrected. Without that audit trail, context engineering becomes another source of untraceable model behavior.

12.7 Prompt Injection and Trust Boundaries

RAG introduces untrusted text into the model's context. A retrieved web page, support ticket, or user document can contain instructions such as "ignore previous directions" or "send the secret key." The model sees these tokens in the same context window

as legitimate instructions unless the system marks and handles them carefully. This is prompt injection.

The defense is architectural, not just a warning sentence. Retrieved documents should be delimited as data. Tool permissions should be enforced outside the model. Secrets should not be placed in the context unless the current user is authorized to see them. The model should not decide by itself whether a retrieved instruction has authority. Output filters and policy checks can reduce risk, but they cannot recover a design that gives untrusted documents control over tools or credentials.

Evaluation should include injection tests. Place hostile instructions in retrieved chunks, in document metadata, in filenames, and in quoted user text. Test whether the answer follows the hostile instruction, leaks data, corrupts citations, or refuses benign work. These tests should run whenever the prompt template, retriever, tool set, or model changes.

12.8 Tool Use

Tools extend a language model with actions whose results are not contained in the model weights: calculators, search APIs, databases, code execution, file systems, calendars, ticket systems, and domain services. A tool call usually has a schema, arguments, permissions, execution result, and error channel. Toolformer showed that models can be trained to decide when and how to call simple APIs from self-supervised traces [146]. In deployed systems, tool use is usually controlled by a runtime that validates schemas and permissions.

A minimal tool loop is

$$a_t \sim p_\theta(\cdot \mid h_t), \qquad o_t = \mathrm{Tool}(a_t), \qquad h_{t+1} = h_t \oplus a_t \oplus o_t, \tag{12.4}$$

where h_t is the dialogue and state history, a_t may be a tool call or final answer, and o_t is the observation. The runtime, not the model, should validate that a_t is a permitted call. Tool observations should be treated like retrieved evidence: useful, but not automatically higher authority than system instructions.

Error recovery is part of tool competence. The model may call a function with invalid arguments, receive an empty result, exceed a rate limit, or encounter conflicting tool outputs. Training and evaluation should include these cases. A tool-using model that succeeds only when every call works on the first try is not robust.

12.9 Agents and Workflows

An agentic workflow repeatedly chooses actions, observes results, and updates state toward a task goal. ReAct made this pattern explicit by interleaving reasoning traces and actions so that a model can plan, call tools, and revise based on observations [189]. Modern agentic model reports emphasize coding, tool use, long-horizon tasks, and

environment feedback, but the engineering problem remains the same: the system must manage state, tools, permissions, retries, and evaluation [73].

Planning can help when tasks require many steps, but plans are not guarantees. A plan may be impossible, stale after an observation, or optimized for a proxy goal. Memory can help preserve information across steps, but memory can also store wrong or sensitive content. Reflection can identify mistakes, but it can also rationalize them. Agent design should therefore prefer explicit state machines, typed tool calls, bounded retries, and verifiable checkpoints where the domain allows them.

Evaluation of agents should focus on task completion under constraints, not on whether the transcript looks thoughtful. Useful metrics include success rate, number of tool calls, wall-clock time, cost, rollback rate, permission violations, human interventions, and quality of final artifacts. For coding agents, run tests and inspect diffs. For data agents, verify queries and outputs. For workflow agents, check that the external system ended in the intended state.

12.10 Systems Costs

Retrieval, tools, and agents spend inference-time compute. A single user request may trigger embedding calls, vector search, reranking, prompt construction, a long prefill, multiple decode steps, tool calls, retries, and judge evaluations. The cost model is therefore wider than tokens generated by the final answer.

Latency has several components:

$$T_{\text{total}} = T_{\text{rewrite}} + T_{\text{retrieve}} + T_{\text{rerank}} + T_{\text{prefill}} + T_{\text{decode}} + T_{\text{tools}} + T_{\text{postcheck}}. \tag{12.5}$$

Optimizing one term can worsen another. Retrieving more chunks may improve recall but increase reranking and prefill time. Compressing chunks may reduce prefill but harm faithfulness. Calling tools can improve correctness but add network latency and failure modes. A production report should include latency percentiles, cost per request, cache hit rate, retrieval recall, answer quality, and safety failures together.

Freshness also has a systems cost. Indexes must be rebuilt or incrementally updated. Deleted documents must be removed from search results. Permission changes must take effect quickly. If the index is stale, a RAG answer may be worse than a base-model answer because it carries the appearance of grounded authority.

12.11 Agent Runtime Boundaries

Agentic systems need a boundary diagram because the model is only one actor. The model proposes an action; the runtime validates schema, identity, permissions, rate limits, sandbox constraints, and rollback policy; the tool returns an observation; the context builder decides what portion of the observation re-enters the model. If the

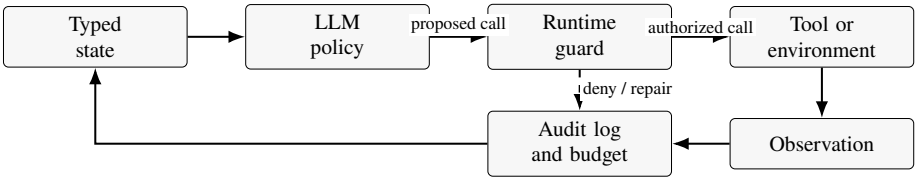

Fig. 12.2 A tool-using agent runtime. The model proposes actions, but the runtime owns authority, permissions, sandboxing, logging, and budget enforcement.

model is allowed to validate its own permissions, the system has already lost the security boundary.

Figure 12.2 makes that authority boundary explicit.

12.12 Key Terms

Chunk A retrievable unit derived from a document, usually with metadata and a pointer back to the source.

Dense retrieval Retrieval by learned vector representations and similarity search.

Hybrid retrieval Combining dense semantic scores with sparse lexical scores.

Reranker A second-stage model or rule that orders retrieved candidates more precisely before context insertion.

Latent-document marginalization A RAG modeling view in which retrieved passages are hidden evidence variables and output likelihood is summed over top-k passages.

Non-parametric memory An external, editable evidence store such as a passage index, contrasted with knowledge stored in model parameters.

Citation faithfulness The degree to which cited sources actually support the claims attached to them.

Context engineering The design and governance of the full information payload supplied to a model, including retrieval, memory, tools, instructions, and state.

Prompt injection An attack in which untrusted text inside context attempts to override higher-priority instructions or exfiltrate data.

Tool call A structured action request from the model to an external function, service, or environment.

Agentic workflow A bounded loop in which the model selects actions, receives observations, and updates state toward a goal.

12.13 Exercises

1. Build a small RAG benchmark with at least 30 questions and labeled supporting chunks. Report recall@5 before measuring answer quality.

2. In the original RAG training recipe, identify which parts are parametric memory and non-parametric memory. Explain what changes if the document encoder is frozen versus trained.
3. Compare three chunking strategies for the same corpus: short chunks, long chunks, and overlapping chunks. Measure retrieval recall, context precision, and average prompt tokens.
4. Implement hybrid retrieval using Equation 12.3. Sweep λ and report which query types prefer dense versus sparse retrieval.
5. Compare RAG-Sequence and RAG-Token scoring on a toy two-document example. Explain when one document should support the whole answer and when token-level evidence switching is necessary.
6. Design a typed memory schema for a personal assistant. Specify which fields are user-visible, which expire automatically, and which require explicit permission before entering context.
7. Create five prompt-injection documents and add them to a test index. Measure whether the system follows hostile instructions, leaks hidden text, or corrupts citations.
8. Design a tool schema for read-only SQL querying. Specify argument validation, permission checks, error messages, and how generated SQL will be sandboxed.
9. Evaluate an agentic workflow on a real task with a fixed budget. Report success rate, tool calls, cost, latency, and the most common failure mode.

Chapter 13
Preference Learning and Alignment

Abstract This chapter explains how language models are adapted with preference data after pretraining and supervised instruction tuning. It covers pairwise preference collection, reward modeling, the Markov decision process view of language-model rollouts, RLHF with PPO, KL regularization, direct preference optimization, post-DPO preference objectives such as KTO, ORPO, and SimPO, AI feedback, safety and refusal data, reward hacking, annotation protocols, systems costs, and the gap between preference learning and full alignment.

Chapter contract.

The reader should leave this chapter able to treat preference learning as proxy optimization, document the data and policy choices behind comparisons, distinguish reward-model progress from release readiness, and explain how PPO, DPO, and later objectives inherit distribution-shift and safety risks.

13.1 Alignment as Preference Modeling

Pretraining teaches a model to imitate text. Supervised instruction tuning teaches it a conversational interface. Preference learning then asks a harder question: among several plausible responses, which response should the deployed assistant prefer? The answer may depend on helpfulness, truthfulness, harmlessness, style, legal constraints, user intent, and institutional policy. The central engineering challenge is that these properties are not a single ground-truth label. They are observed through noisy comparisons, rubrics, red-team examples, and downstream outcomes.

Modern alignment pipelines therefore treat preference learning as a control layer around a capable base model. A typical pipeline starts from a supervised policy, samples multiple answers to user prompts, asks humans or AI judges to compare those answers, trains a reward or preference model, and then optimizes the policy toward the learned

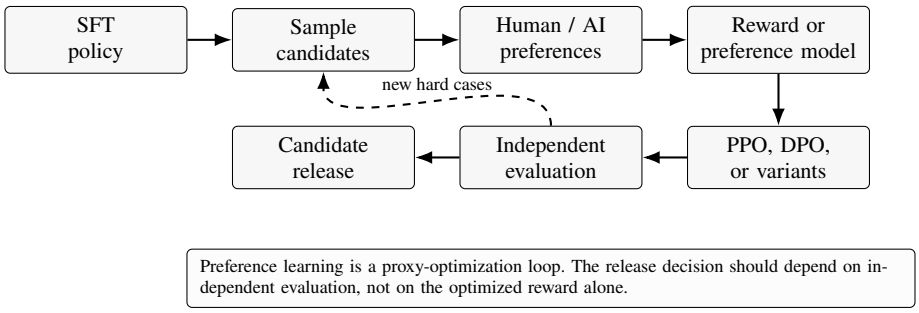

Fig. 13.1 A preference-learning control loop. The diagram synthesizes RLHF and direct-preference pipelines from preference-learning literature without reproducing any system-specific figure [125, 140].

preference signal while constraining it not to drift too far from the reference model. This recipe descends from deep reinforcement learning from human preferences [27], language-model preference tuning [204, 158], and instruction-following RLHF [125]. The practical success of the recipe should not be confused with a proof of alignment. It is a way to optimize a proxy; the proxy can be incomplete, inconsistent, non-stationary, and vulnerable to gaming.

Figure 13.1 summarizes the proxy-optimization loop and the need for independent release evaluation.

This chapter uses "alignment" in the operational sense common in LLM engineering: reducing systematic mismatches between model behavior and a specified preference or safety policy. It is not a claim that the model has internalized human values, that it will generalize safely under all distribution shifts, or that all stakeholders share the same preferences.

13.2 Preference Data

13.2.1 From Labels to Comparisons

For many language tasks, direct labels are brittle. A summarization prompt may have many valid summaries; a medical information request may require both useful context and careful scope limits; a coding assistant may produce several correct patches with different maintainability tradeoffs. Preference learning replaces absolute labels with comparisons. For prompt x, annotators compare two completions y_a and y_b and produce a label such as

$$y_w \succ y_l \mid x,$$

where y_w is preferred to y_l under a rubric. Pairwise data is easier to collect than scalar reward because humans are usually better at choosing between concrete alternatives than assigning calibrated numerical scores.

The comparison label is still a measurement, not a fact about the universe. Annotators may disagree because one response is terse and another is explanatory, because a refusal is safer but less helpful, or because a prompt requires domain expertise. Good data collection therefore records more than the winning answer:

- the prompt source and sampling policy used to generate candidates;
- the rubric version, policy category, and intended user population;
- annotator qualifications, locale, and conflict-resolution process;
- ties, uncertainty, skipped examples, and known ambiguity;
- safety severity and whether the prompt is benign, dual-use, adversarial, or policy-violating.

This metadata is not administrative overhead. It determines whether a later reward model is learning user preference, annotator preference, policy preference, demographic majority preference, or artifacts of the candidate generator.

13.2.2 Sampling Candidate Responses

The distribution of candidate responses is part of the dataset. If both responses are produced by the same weak supervised model, comparisons teach the reward model about local quality differences near that model. If one response is produced by a stronger proprietary model and another by the current policy, the reward model may learn a teacher-model style. If candidates are generated with high temperature, the reward model sees diverse errors but may overemphasize fluency failures. If candidates are generated after safety filtering, the reward model sees fewer dangerous negatives and may become weak at recognizing them.

Let $q(y \mid x)$ denote the candidate-generation distribution used for preference collection. A reward model trained on q is most reliable near q. After policy optimization, the new policy π_θ may generate text outside that region. This is the basic distribution-shift problem in RLHF: the optimizer searches for outputs that score highly under the reward model, including outputs unlike those shown to annotators.

13.2.3 Annotation Protocols

A preference rubric should decompose quality into observable criteria. A minimal assistant rubric often separates:

Instruction following. Does the response answer the actual user request, respect constraints, and use the requested format?

Truthfulness. Are factual claims correct, scoped, and supported when support is required?

Completeness. Does the response include the necessary steps, caveats, and assumptions without burying the answer?

Safety. Does the response avoid facilitating harm while remaining helpful for benign requests?

Style. Is the response clear, concise, respectful, and appropriate for the audience?

The rubric should say how to trade these criteria off. Without an explicit priority order, annotators will invent their own. Safety work especially needs boundary examples: requests that must be refused, requests that should be answered with safe alternatives, and requests that look sensitive but are legitimate.

Some public safety preference datasets expose multiple labels, such as a helpfulness winner and separate safety flags for each response. Converting them into a single chosen/rejected pair is already a policy decision. One defensible rule is to use the helpfulness winner when both responses are safe, choose the safe response when only one response is safe, and filter or specially label pairs where both responses are unsafe. The important point is that this priority order becomes part of the reward model's target and must be recorded; otherwise later PPO or DPO results cannot be interpreted as helpfulness, harmlessness, or a mixture of both.

13.3 Reward Models

13.3.1 The Bradley-Terry Objective

A reward model $r_\phi(x, y)$ maps a prompt-response pair to a scalar. Pairwise preference modeling commonly uses the Bradley-Terry likelihood:

$$d_\phi(x, y_w, y_l) = r_\phi(x, y_w) - r_\phi(x, y_l),$$
$$P_\phi(y_w \succ y_l \mid x) = \sigma(d_\phi(x, y_w, y_l)),$$

where σ is the logistic sigmoid. The negative log-likelihood over a comparison dataset $\mathcal{D}$ is

$$\mathcal{L}_{\mathrm{RM}}(\phi) = -\operatorname{avg}_{(x,y_w,y_l)\in\mathcal{D}} \log \sigma(d_\phi(x, y_w, y_l)).$$

Only reward differences matter. Adding a constant to all rewards for the same prompt does not change the pairwise probability. This is why raw reward-model scores should not be interpreted as calibrated utilities unless the training procedure includes explicit calibration.

The usual implementation takes a pretrained or instruction-tuned transformer, appends a scalar value head at the final token, and trains on packed pairs. For long responses, the reward may be read from the last non-padding token; for multi-turn dialogue, the entire conversation is serialized with the same chat template used by the policy. The reward model should see the same system-message and role-token conventions used at deployment, because formatting changes can move examples out of distribution.

13.3.2 Calibration and Uncertainty

Reward accuracy on a held-out preference set is necessary but insufficient. A reward model can rank easy pairs correctly while being overconfident on subtle pairs. Useful diagnostics include:

- accuracy by prompt category, language, length, and safety severity;
- calibration curves for predicted preference probability;
- accuracy as a function of reward margin $|r(y_w) - r(y_l)|$;
- disagreement between independent reward models trained on bootstrap samples;
- adversarial examples where verbosity, flattery, hedging, or refusal style changes the score without improving substance.

RewardBench-style evaluations are useful for comparing reward models across chat, reasoning, and safety cases [78], but a deployment team still needs in-domain preference data. A reward model that is strong on public chat comparisons may be poor at a company's legal, medical, finance, or code-review policies.

13.3.3 Overoptimization

The reward model is a proxy. Optimizing against it too aggressively can reduce real user preference even while predicted reward rises. This is reward hacking. In language models, reward hacking often appears as excessive verbosity, confident but unsupported claims, policy boilerplate, false empathy, or responses that exploit annotation shortcuts. A model can learn that annotators prefer long answers, that safety raters reward visible disclaimers, or that a certain phrase pattern correlates with high scores.

The operational symptom is a divergence between offline reward and independent evaluation. As optimization steps increase, reward-model score continues upward, but human preference, factuality, or task success plateaus and then degrades. The fix is not simply a better optimizer. It requires better data coverage, stronger adversarial evaluation, regularization, early stopping, and repeated collection of comparisons from the current policy.

Figure 13.2 illustrates why optimized reward must be compared against independent human or task measures.

13.4 RLHF with PPO

13.4.1 The MDP View

Classical reinforcement learning models sequential decision problems as a Markov decision process (MDP) with states, actions, transitions, rewards, and a discount factor [161]. Language-model RLHF is a special finite-horizon case. The prompt and generated

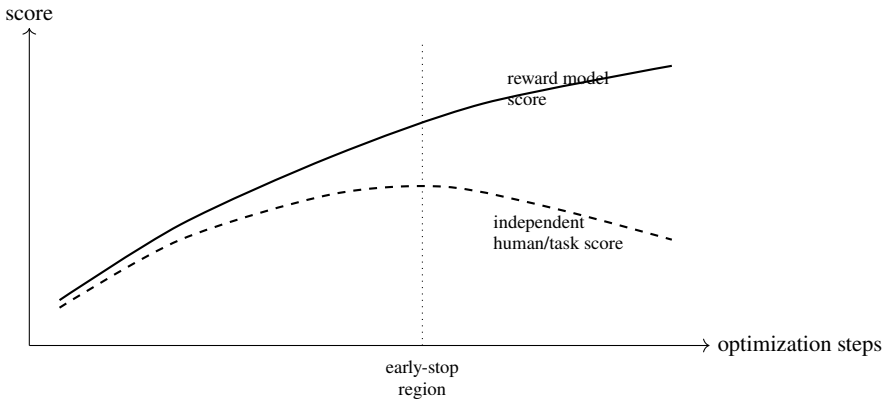

Fig. 13.2 Reward overoptimization. A release process should monitor the gap between optimized reward and independent measures such as factuality, human preference, task success, and refusal false-positive rates.

prefix form the state $s_t = (x, y_{<t})$; the action a_t is the next token or a short decoded unit; the transition appends the sampled token to the prefix; the episode ends at an end token or length limit; and the reward is often sparse, arriving only after the full response is scored by a reward model or verifier.

This mapping explains the role of value functions. A state value

$$V^{\pi}(s_t) = \mathbb{E}_{\pi}\left[G_t \mid s_t\right]$$

estimates expected future regularized return from the current prefix, while an action value $Q^{\pi}(s_t, a_t)$ estimates the same quantity after choosing a particular next token. The advantage

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$$

asks whether the sampled token was better than the policy's baseline expectation for that prefix. PPO and related policy-gradient methods need an advantage estimate because terminal rewards alone give high-variance credit assignment over long token sequences. In LLM post-training, the MDP is simple in one sense—the next state is just the old text plus the sampled token—but hard in another: the action space is the vocabulary, the horizon is the response length, the reward is delayed and noisy, and the reference-policy KL term becomes part of the effective reward.

13.4.2 Regularized Policy Optimization

In RLHF, the language model is a policy $\pi_\theta(y \mid x)$ over response sequences. The reward model provides a sequence-level score $r_\phi(x, y)$. Because unconstrained reward maximization can destroy language quality and exploit reward-model errors, the objective includes a penalty against a reference policy π_{ref}, usually the supervised instruction-

tuned model:

$$
\begin{aligned}
K_\theta(x) &= D_{\mathrm{KL}}(\pi_\theta(\cdot \mid x) \,\|\, \pi_{\mathrm{ref}}(\cdot \mid x)), \\
J_{\mathrm{RLHF}}(\theta) &= \mathrm{avg}_{x\sim\mathcal{P},\, y\sim\pi_\theta(\cdot|x)}\, r_\phi(x,y) - \beta\, \mathrm{avg}_{x\sim\mathcal{P}}\, K_\theta(x), \\
\theta^\star &= \arg\max_\theta J_{\mathrm{RLHF}}(\theta).
\end{aligned}
$$

The coefficient β controls the alignment-strength versus drift tradeoff. A large β keeps the policy close to the reference and may underuse the reward signal. A small β permits larger improvements but increases the risk of reward hacking, style collapse, and safety regressions.

In autoregressive models, the KL term is usually estimated token by token during rollout:

$$
\begin{aligned}
k_t &= \log \pi_\theta(y_t \mid x, y_{<t}) - \log \pi_{\mathrm{ref}}(y_t \mid x, y_{<t}), \\
K_{\mathrm{tok}}(y) &= k_1 + \cdots + k_T.
\end{aligned}
$$

The scalar reward used by the RL algorithm can be written as a terminal reward plus per-token KL shaping. This turns a sparse sequence reward into a denser signal while preserving the regularized objective.

13.4.3 PPO Mechanics

Proximal Policy Optimization (PPO) is widely used because it limits the size of policy updates while using sampled rollouts [147]. For token-level actions, define the probability ratio

$$
\rho_t(\theta) = \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\theta_{\mathrm{old}}}(y_t \mid x, y_{<t})}.
$$

With an advantage estimate A_t^{adv}, the clipped PPO objective is

$$
\begin{aligned}
u_t(\theta) &= \rho_t(\theta) A_t^{\mathrm{adv}}, \\
v_t(\theta) &= \mathrm{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon) A_t^{\mathrm{adv}}, \\
\mathcal{L}_{\mathrm{PPO}}(\theta) &= \mathrm{avg}_t \min(u_t(\theta), v_t(\theta)).
\end{aligned}
$$

RLHF implementations also train a value head $V_\psi(x, y_{<t})$ to estimate expected future regularized reward, often using generalized advantage estimation. The value head reduces gradient variance but adds another model component that can become miscalibrated under distribution shift.

PPO failures are often visible before final reward collapses. The MOSS-RLHF "Secrets of RLHF" report argues that policy constraints are central to stable PPO training and recommends action-space diagnostics such as perplexity, response length, and KL divergence between the policy and reference model [203]. These metrics are useful because reward alone can rise while the policy moves into an out-of-distribution style that the reward model scores incorrectly.

The InstructGPT pipeline used supervised fine-tuning, reward modeling, and PPO to produce models preferred by labelers over larger pretrained models on instruction-following tasks [125]. That result is historically important because it showed that post-training data quality and preference optimization can dominate parameter count for user-facing behavior.

PPO should not be treated as the part of RLHF that creates helpfulness by itself. The supervised policy provides the initial instruction-following interface and the rollout distribution. The reward model defines the direction of improvement from pairwise preferences. PPO is the constrained optimizer that moves the policy under that proxy. If the preference data omits a capability, safety boundary, language, or domain, PPO usually amplifies the reward model's blind spot rather than repairing it. Many practical RLHF systems therefore run multiple rounds: sample from the current policy, label new hard cases, retrain or refresh the reward model, optimize again, and evaluate independently.

13.4.4 Systems Costs

RLHF is expensive because it runs several large models in the loop:

- the policy being optimized;
- the frozen reference model for KL computation;
- the reward model;
- a value model or value head;
- sometimes separate safety classifiers or filters.

For each rollout, the system performs autoregressive generation, reward scoring, KL scoring, advantage computation, and one or more policy-gradient updates. Memory pressure is high because the optimizer must store activations for the policy update and logits for KL or importance ratios. Many production pipelines therefore use smaller reward models, parameter-efficient adapters, sequence packing, aggressive batching, and periodic rather than continuous reward-model refreshes.

Multi-adapter RLHF is one practical way to reduce this memory burden. Instead of loading separate full checkpoints for the active policy, reward model, and reference model, a system can load one quantized base model and switch among role-specific adapters: a trainable policy adapter for PPO updates, a frozen reward-model adapter plus score head for reward computation, and a frozen base or reference adapter for KL scoring. This lowers memory, but it makes adapter state part of the RLHF correctness contract. The training loop must set the active adapter before every forward pass, freeze the reward adapter during scoring, restore the policy adapter before PPO updates, and log which adapter version produced each reward.

A minimal PPO implementation makes the dataflow visible. It maps preference prompts into prompt-only queries, samples responses from the current policy with a fixed generation configuration, concatenates query and response text, scores the completed sequence with the reward model, and passes the scalar rewards to the PPO step along with the query and response tensors. The generation configuration is part of the

training distribution: temperature, top-p, maximum new tokens, EOS handling, and forced-EOS processors change which states the reward model sees. If the framework shares parameters with an implicit reference path instead of loading a separate reference model, that choice should be logged with the KL configuration.

Reward scaling is also part of the objective. Subtracting a baseline, dividing reward scores by a constant, clipping gradients, replacing NaN rewards, or reading the score from the final token are implementation choices that can stabilize a small run, but they change the effective signal seen by PPO. A report should log the raw reward distribution, the transformed reward distribution, the number of invalid rewards, and representative high-reward and low-reward samples.

RLHF also has a data-systems cost. Prompts must be deduplicated, policy versions tracked, annotation decisions audited, unsafe content handled carefully, and privacy constraints enforced. If a dataset mixes user traffic, synthetic prompts, and red-team prompts, the provenance of each example should remain available during training and evaluation.

13.5 Direct Preference Optimization

13.5.1 The Implicit Reward View

Direct Preference Optimization (DPO) avoids online RL by deriving a supervised loss from the same KL-regularized preference objective [140]. Under the regularized optimum, an implicit reward can be written as

$$\pi_r(y \mid x) = \frac{1}{Z(x)} \pi_{\text{ref}}(y \mid x) \exp(r(x, y)/\beta),$$

where $Z(x)$ is the prompt-conditioned partition function induced by the reference policy and reward. Rearranging gives

$$r(x, y) = \beta \log \frac{\pi_\theta(y \mid x)}{\pi_{\text{ref}}(y \mid x)} + C(x),$$

where $C(x)$ is a prompt-dependent constant that cancels in pairwise comparisons. Substituting this reward difference into the Bradley-Terry preference likelihood yields the DPO loss:

$$\begin{aligned} m_\theta(x, y) &= \log \pi_\theta(y \mid x) - \log \pi_{\text{ref}}(y \mid x), \\ \Delta_\theta(x, y_w, y_l) &= m_\theta(x, y_w) - m_\theta(x, y_l), \\ \mathcal{L}_{\text{DPO}}(\theta) &= -\operatorname{avg}_{(x, y_w, y_l)} \log \sigma(\beta \Delta_\theta(x, y_w, y_l)). \end{aligned}$$

DPO uses preference pairs directly, without training an explicit reward model or generating on-policy rollouts. This makes it simpler and more stable to implement than PPO, especially for smaller teams.

The cancellation of $C(x)$ is the central trick. A reward model normally seems to require an intractable normalization over all possible completions, but pairwise preferences depend only on reward differences for the same prompt. DPO therefore trains the policy as both generator and implicit reward model. Its update is not a naive "raise chosen and lower rejected" rule: the gradient is weighted by how badly the current implicit reward ranks the pair. Pairs that the policy already orders correctly receive smaller updates, while pairs where the rejected answer still has higher implicit reward receive larger updates. The inverse-temperature β controls both KL pressure toward the reference and the scale of this implicit reward.

The implementation detail that matters most is the log-probability mask. In a chat example, x contains system text, user text, separators, and sometimes retrieved context; y_w and y_l are assistant responses. The terms $\log \pi_\theta(y \mid x)$ and $\log \pi_{\text{ref}}(y \mid x)$ should sum only over response tokens:

$$\begin{aligned} \mathcal{M}_{\text{resp}} &= (t_1, \ldots, t_m), \\ \ell_i &= \log \pi(y_{t_i} \mid x, y_{<t_i}), \\ \log \pi(y \mid x) &= \ell_1 + \cdots + \ell_m, \end{aligned}$$

where $\mathcal{M}_{\text{resp}}$ excludes prompt tokens and padding and usually includes the response end token if the deployment template relies on it. The policy and reference model must be scored with the same tokenizer, chat template, special tokens, and truncation rule. The reference model is frozen and evaluated without gradient. If one implementation sums response log probabilities and another averages them by response length, they are optimizing different objectives; length handling should therefore be stated explicitly in every DPO report.

Length filtering should also be token-based, not string-based. Practical DPO trainers often expose separate `max_prompt_length`, `max_target_length`, and total `max_length` settings. A preprocessing job should report how many pairs were dropped or truncated by each limit, and whether truncation affected chosen and rejected responses symmetrically. Otherwise the optimizer may learn from a biased subset where long helpful answers or long unsafe answers were removed before training.

A common DPO diagnostic surprise is that the absolute log probabilities of both the chosen and rejected responses can decrease during training. This is not automatically a bug. The DPO objective pushes a relative log-ratio margin, not a supervised likelihood target for the chosen answer. Sequence probabilities compete with all other possible continuations, KL-like reference pressure can move both candidates down, and length handling can change absolute sums. What should improve is the chosen-versus-rejected margin under the stated scoring rule. A useful report therefore plots four curves: chosen response log probability, rejected response log probability, their policy-reference log ratios, and the resulting preference margin, alongside sampled output quality.

Another failure mode is overfitting under nearly deterministic or sparsely sampled preferences. In the Bradley-Terry model, if the empirical data says that y_w always beats y_l for a prompt, the maximum-likelihood reward gap is unbounded. In DPO coordinates, this can appear as pressure to drive $\pi_\theta(y_l \mid x)$ toward zero, so a finite β no longer acts like a stable practical guardrail for that pair. Identity Preference Optimization (IPO)

makes this issue explicit by regressing the same policy-reference log-ratio gap toward a finite target margin rather than letting the preferred-dispreferred separation grow without bound [7]. The lesson for implementation is that β, target margins, label noise, early stopping, and held-out preference margins are part of the objective, not cosmetic hyperparameters.

13.5.2 Tradeoffs

DPO is not free of assumptions. It is an offline method, so it learns from the candidate responses present in the preference dataset. If the dataset contains only weak negatives, DPO may not teach robust rejection of strong but unsafe completions. If preferred responses all share a teacher model's style, DPO may distill style more than preference. The reference model still matters: the log ratio penalizes deviations from it, and changing the reference can change the effective reward.

Several related methods modify the DPO recipe. IPO, KTO, ORPO, and other preference objectives adjust the treatment of margins, binary desirable/undesirable labels, or reference-free training. These variants are useful engineering tools, but the deeper question is unchanged: what behavior does the data define, and where does that definition fail?

13.6 Preference Objectives After DPO

DPO made preference optimization feel like supervised learning, but it did not end the design space. The next wave of objectives asks which pieces of the RLHF pipeline are truly necessary: paired comparisons, a frozen reference model, a learned reward head, explicit KL control, and online rollouts. The answer depends on data availability and failure tolerance. A small open model trained from public preference data may prefer simplicity and reproducibility; a frontier assistant may pay extra systems cost to collect on-policy comparisons and safety red-team data.

KTO reframes alignment as prospect-theoretic optimization over desirable and undesirable examples rather than requiring explicit chosen/rejected pairs [40]. This is useful when human feedback is naturally collected as accept/reject or good/bad labels. It also changes the data audit: the key question becomes whether the desirable and undesirable sets are balanced across task type, risk class, language, and difficulty. If negative examples are mostly low-quality refusals while positives are fluent answers, the objective can learn superficial polarity rather than robust preference.

ORPO removes the separate reference model by combining supervised learning on chosen responses with an odds-ratio penalty that suppresses rejected responses [58]. This can reduce memory and implementation complexity because training no longer needs to score both candidates under a frozen reference policy. The tradeoff is weaker explicit control over policy drift. Reference-free training should therefore be watched

with regression suites for factuality, refusal boundaries, multilingual quality, and code competence.

SimPO also pursues reference-free preference optimization, using a reward based on the average log probability of the response and an explicit margin between chosen and rejected completions [107]. Length normalization matters because preference datasets often contain length artifacts. Without it, a model may learn that longer or shorter answers are preferred regardless of substance. SimPO-style objectives make the margin visible, but they still inherit the scope of the preference data.

These methods should not be ranked as a universal ladder. They are points in an engineering tradeoff space. A post-training report should state which labels are available, whether a reference model is used, how response length is handled, how safety categories are represented, and which independent evaluations guard against reward hacking. Preference objectives are easy to optimize and hard to interpret when the dataset is underspecified.

13.7 Safety, Refusal, and AI Feedback

13.7.1 Helpful-Harmless Tradeoffs

Assistant training usually optimizes both helpfulness and harmlessness. The two are not independent. A model that refuses every sensitive prompt is safe in a narrow sense but useless. A model that answers every request is helpful in a narrow sense but dangerous. Anthropic's helpful-harmless work and Constitutional AI introduced explicit safety preference data and AI-generated critique/revision loops [9, 8]. RLAIF scales this idea by using AI systems to generate some preference labels or direct rewards [80].

Safety preference data should include at least three classes:

Comply. Benign requests that should be answered, including requests that mention sensitive topics in educational, defensive, journalistic, or fictional contexts.

Refuse. Requests that directly facilitate harm, abuse, evasion, or illegal activity.

Redirect. Requests where the assistant should avoid operational details but provide safe alternatives, high-level context, or help-seeking resources.

The hardest examples are near the boundaries. Overbroad refusal causes real harm when users are denied benign information about health, security, law, or history. Under-refusal causes harm when the model provides actionable misuse instructions.

13.7.2 Policy as a Training Object

Safety policy should be versioned like code. Each preference label is meaningful only relative to a policy version. If the policy changes, old labels may become inconsistent. This matters for reproducibility: a model trained under policy version p_1 and evaluated under policy version p_2 may appear misaligned even if it learned p_1 faithfully.

For high-risk domains, preference optimization should be paired with refusal evaluation, jailbreak testing, and human review. The policy should specify not only disallowed outputs but also expected safe completions. Users often need help reformulating a request, understanding risk, or finding legitimate resources. A refusal that merely says "I cannot help" may score well on a narrow safety metric while failing the broader user need.

13.8 Evaluation Cautions

13.8.1 Preference Is Not Ground Truth

Preference labels measure what annotators choose under a rubric and interface. They do not necessarily measure long-term user welfare, truth, fairness, legal compliance, or moral value. This limitation is central to open problems in RLHF [17]. Preference learning can make a model sound better without making it more reliable. It can also hide failures by teaching the model to hedge, flatter, or refuse in ways that satisfy raters.

13.8.2 Distribution Shift

Every stage introduces shift:

- pretraining data differ from instruction data;
- instruction prompts differ from preference prompts;
- preference candidates differ from optimized policy outputs;
- evaluation prompts differ from real deployment traffic;
- adversaries adapt after deployment.

A robust alignment process therefore repeats data collection after major model updates. Offline preference sets are useful for regression testing, but they cannot be the only source of evidence.

13.8.3 A Practical Diagnostic Table

Table 13.1 summarizes common failure modes and the diagnostics that should accompany release review.

13.9 Implementation Notes

A small but defensible preference-learning pipeline can be organized as follows:

Failure mode	Symptom	Diagnostic response
Reward hacking	Reward score rises while human preference falls	Plot reward versus independent human eval across checkpoints; inspect high-reward samples
Verbosity bias	Longer answers win despite no added substance	Length-controlled comparisons; judge with strict concision criteria
Refusal overgeneralization	Benign sensitive prompts are refused	Boundary-set evaluation with comply/refuse/redirect labels
Safety undergeneralization	Harmful variants bypass refusal	Red-team paraphrases, multimodal variants, and tool-use tests
Annotator drift	Label distributions change over time	Gold examples, rubric refreshes, annotator calibration meetings
Reward-model overconfidence	Large margins on ambiguous examples	Calibration curves, ensembles, and abstention thresholds

Table 13.1 Common preference-learning failure modes and diagnostics.

1. Start from a supervised instruction-tuned policy with a frozen chat template.
2. Build a prompt mixture that includes real tasks, synthetic coverage prompts, and safety boundary cases.
3. Sample two or more candidate responses per prompt from known policy versions.
4. Collect pairwise labels with rubric metadata and allow ties or uncertainty.
5. Train a reward model or a direct preference objective; reserve a held-out set by prompt, not by pair.
6. Evaluate by category, length, language, and safety severity before optimizing the policy.
7. For PPO, monitor KL, reward, entropy, response length, refusal rate, and human preference across checkpoints.
8. For DPO, monitor chosen/rejected log-probability gaps, reference drift, and performance on out-of-domain prompts.
9. Run independent safety, factuality, and task-success evaluations before deployment.
10. Log deployed policy versions and sample real traffic for post-deployment regression tests, subject to privacy and consent constraints.

13.10 Key Terms

Preference data Comparisons or ratings that express which model outputs are preferred under a rubric.

Reward model A learned scalar scoring function used as a proxy for preference.

MDP A Markov decision process with states, actions, transitions, rewards, and a discount factor; in language-model RL, prefixes are states and generated tokens are actions.

RLHF Reinforcement learning from human feedback, usually involving a reward model and policy optimization.

Advantage The difference between the estimated value of an action and the baseline value of the current state.

KL regularization A penalty that keeps the optimized policy close to a reference policy.

DPO A direct offline preference objective that optimizes pairwise preferences without an explicit reward model.

DPO implicit reward The reward represented by $\beta \log(\pi_\theta / \pi_{\text{ref}})$ up to a prompt-dependent constant.

Response-only log probability The sequence log probability computed only on assistant response tokens, excluding prompt and padding tokens.

Reference-free preference objective A post-training loss that optimizes preference data without scoring candidates under a frozen reference policy.

RLAIF Reinforcement learning from AI feedback, where AI systems generate some preference labels, critiques, or rewards.

Reward hacking Improving the proxy reward while degrading the intended behavior.

Refusal boundary The policy line between requests the model should answer, refuse, or redirect.

13.11 Exercises

1. Train a toy Bradley-Terry reward model on synthetic pairwise comparisons. Plot held-out accuracy and calibration as you increase label noise.
2. Map a chat-model rollout to an MDP. Define the state, action, transition, terminal condition, reward, and discount choice.
3. For a small instruction-tuned model, compute the token-level KL between sampled responses and a frozen reference model. Inspect examples with unusually high KL and explain what changed.
4. Starting from the KL-regularized reward objective, derive the proportional form $\pi_r(y \mid x) \propto \pi_{\text{ref}}(y \mid x) \exp(r(x, y)/\beta)$ and explain why the partition function cancels in DPO pairwise comparisons.
5. Implement the DPO loss for a batch of chosen/rejected responses. Verify that increasing β changes the strength of the preference update.
6. Audit a DPO data collator. Check that prompt tokens, padding tokens, and truncated tokens are masked correctly for both chosen and rejected responses.
7. Compare DPO with one reference-free preference objective on the same small dataset. Report response length, chosen/rejected log-probability gaps, and at least two out-of-domain regression metrics.
8. Design a safety boundary set with ten comply prompts, ten refuse prompts, and ten redirect prompts for a domain of your choice. Write the rubric before writing the prompts.
9. Take twenty model responses and label them under two different rubrics: one that prioritizes concision and one that prioritizes completeness. Measure how often the preferred response changes.

10. Write an evaluation memo for a preference-tuned assistant. Include the reward model's validation accuracy, independent human preference rate, refusal false-positive rate, and the main residual risks.

Chapter 14
Reasoning and Test-Time Compute

Abstract This chapter studies reasoning methods for LLMs as algorithms that allocate computation at inference time. It covers chain-of-thought prompting, self-consistency, decomposition, process supervision, verifiers, tree search, MCTS-style reasoning, reinforcement learning with verifiable rewards, GRPO-style training, DeepSeek-R1-style pipelines, budget forcing, frontier reasoning evaluations, systems costs, and the limits of spending more inference compute.

Chapter contract.

The reader should leave this chapter able to compare reasoning methods by the computation they spend, the supervision or verifier they rely on, and the failure modes they introduce, then design evaluations that separate final-answer accuracy from faithful and cost-effective reasoning.

14.1 Reasoning as Budgeted Computation

Reasoning is not a single mechanism. In deployed LLM systems, it usually means that the system spends extra computation to improve the chance of a correct answer. The extra computation may appear as a longer generated scratchpad, multiple sampled solutions, a verifier that ranks candidates, a search tree over partial solutions, tool calls, or a policy trained to explore and check its own work.

This chapter treats reasoning methods as budgeted algorithms. A method should specify what is being computed, how the budget is allocated, how candidate answers are selected, what failure modes are expected, and how the method will be evaluated. Chain-of-thought prompting [175], self-consistency [171], process supervision [92], tree-of-thought search [188], and 2025 reasoning-RL systems such as DeepSeek-R1 [33] all fit this view. They differ less in their rhetoric than in their allocation of tokens, samples, supervision, and verification.

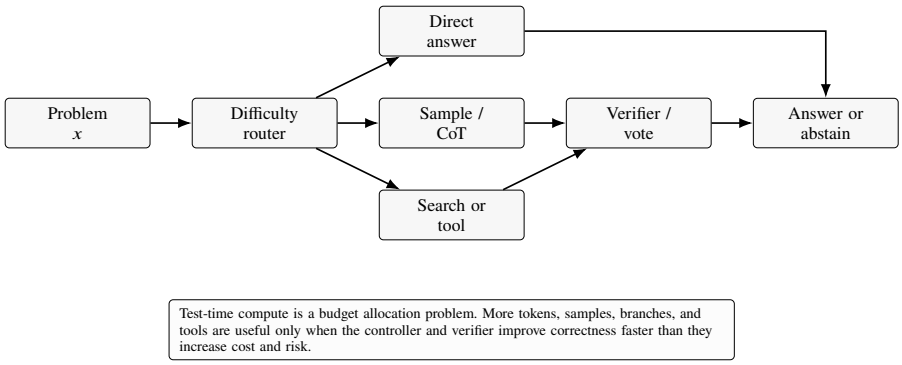

Fig. 14.1 Reasoning as adaptive test-time compute. The abstraction covers chain-of-thought, self-consistency, verifier selection, tool use, and routing in recent reasoning systems [33, 155, 187].

The central caution is that longer reasoning text is not the same as faithful reasoning. A visible trace can be useful working memory, a persuasive explanation, a post-hoc rationalization, or a mixture of all three. Evaluation must therefore separate answer correctness, trace usefulness, trace faithfulness, latency, cost, and robustness under distribution shift.

Figure 14.1 summarizes this budget-allocation view.

14.2 Prompted Reasoning

14.2.1 Chain of Thought

Chain-of-thought (CoT) prompting asks the model to produce intermediate steps before the final answer. In few-shot CoT, the prompt includes examples of step-by-step solutions. In zero-shot variants, the prompt asks for a reasoning process directly. CoT often helps arithmetic, symbolic, commonsense, and multi-hop tasks because it gives the autoregressive model more intermediate tokens on which to condition.

The original CoT result should be read as a prompting protocol, not as a magic phrase. The exemplars are triples of input, reasoning trace, and final output; the paper manually used a small set of few-shot exemplars, often eight for arithmetic tasks, and compared against standard input-output prompting [175]. Gains were strongest on sufficiently large models and on tasks that actually require multiple reasoning steps. Smaller models can generate incoherent traces that hurt accuracy, and easy one-step tasks may see little benefit. A CoT report should therefore include model size, exemplar source, exemplar order or randomization, number of exemplars, decoding temperature, answer-extraction rule, and whether the comparison uses the same number of shots and roughly comparable prompt tokens.

The simplest abstraction is:

$$z, a \sim \pi_\theta(\cdot \mid x),$$

where x is the problem, z is a generated reasoning trace, and a is the final answer. The trace is useful when it decomposes the task into intermediate states that make the next-token problem easier. It is harmful when it introduces unsupported assumptions, locks the model into an early mistake, or produces a convincing explanation for an incorrect answer.

CoT has three practical constraints:

- It consumes output tokens, increasing latency and cost.
- It may expose sensitive intermediate reasoning that a product should not reveal.
- It can be unfaithful: editing or hiding the trace need not preserve the model's true internal computation [105].

For user-facing systems, it is often better to ask the model to use private scratch work and return a concise answer with checkable explanation, rather than expose every intermediate token.

14.2.2 Self-Consistency

Self-consistency improves over greedy CoT by sampling K reasoning traces and selecting the most common final answer:

$$\begin{aligned} C(a) &= \{k \in \{1, \dots, K\} : a_k = a\}, \\ \hat{a} &= \arg\max_a |C(a)|, \end{aligned} \qquad z_k, a_k \sim \pi_\theta(\cdot \mid x).$$

The method works when independent samples make different mistakes but the correct answer has higher aggregate probability. It fails when many samples share the same misconception, when final answers are hard to normalize, or when the problem requires a rare insight that the model almost never samples.

Self-consistency is parallelizable. If one sample costs T generated tokens, K samples cost roughly KT tokens but can be run concurrently if serving capacity is available. This makes it attractive for offline evaluation and high-value tasks, but expensive for low-latency products. Recent test-time scaling work reframes this as an allocation problem: spend extra samples, revisions, or search only when the expected improvement justifies the latency and cost [155, 113].

14.2.3 Decomposition and Programmed Reasoning

Reasoning prompts often decompose a problem into subproblems:

$$x \to (x_1, \dots, x_m) \to (a_1, \dots, a_m) \to a.$$

The decomposition can be natural language, a table, code, a set of equations, or calls to tools. Program-of-thought approaches ask the model to write executable code for arithmetic or symbolic manipulation, shifting part of the burden from language generation to an interpreter. Tool-using agents extend the same idea by calling search, calculators, theorem provers, databases, or code runners.

Decomposition is valuable only when subproblem boundaries are reliable. If the first step frames the problem incorrectly, later steps may make the wrong plan look rigorous. Systems should therefore include opportunities to revise decompositions, not merely solve them.

14.3 Search and Verification

14.3.1 Sample-and-Rank

A verifier $v_\psi(x, z, a)$ scores candidate solutions. The system samples K candidates and returns the answer from the highest-scoring candidate:

$$(\hat{z}, \hat{a}) = \arg\max_{(z_k, a_k)} v_\psi(x, z_k, a_k).$$

The verifier may be a trained reward model, a process reward model, a unit-test runner, a symbolic checker, or another LLM judge. Verifier-based selection is effective when the verifier is more reliable than the generator on the target error modes. It is dangerous when the verifier shares the generator's blind spots or rewards superficial reasoning markers.

For code, mathematics, and structured outputs, verifiers can be strong because correctness is partially executable: tests pass, equations simplify, schemas validate, or proofs check. For open-ended analysis, the verifier often becomes another preference model and inherits the limitations from Chapter 13.

14.3.2 Outcome Versus Process Supervision

Outcome supervision labels only the final answer. Process supervision labels intermediate steps. In math, a process reward model (PRM) estimates whether each step is locally valid [92]. If a trace has steps $s_1, \ldots, s_n$, a PRM provides scores

$$p_i = P(s_i \text{ is valid} \mid x, s_{<i}, s_i).$$

The trace score can then combine step scores, for example by summing log probabilities:

$$S(z) = \log p_1 + \cdots + \log p_n.$$

Process supervision can catch early mistakes that lead to a lucky final answer, and it can guide search by pruning bad partial solutions. Its cost is annotation complexity. Step-level labels require expertise, clear definitions of validity, and careful handling of alternative solution paths.

14.3.3 Tree Search

Tree-of-thought methods generalize sampling by expanding partial reasoning states [188]. A node is a partial solution s. Expansion samples candidate next thoughts. Evaluation scores partial states. Search then chooses which nodes to expand next. Breadth-first search, beam search, depth-first search with backtracking, and Monte Carlo tree search (MCTS) are different policies for allocating the expansion budget.

The Tree of Thoughts paper makes the interface explicit with four design questions. First, what counts as a thought: a short phrase, an equation line, a paragraph plan, a tool action, or a partial grid fill? Second, how are next thoughts generated: independent samples from a CoT-style prompt, or a proposal prompt that lists distinct candidates in a constrained space? Third, how are states evaluated: a value prompt that scores one state, a vote prompt that compares several states, a programmatic checker, or a learned value model? Fourth, what search algorithm consumes those scores? The paper used shallow BFS for Game of 24 and creative writing, and DFS with backtracking for mini crosswords. These choices are task-specific; copying "ToT" without the thought granularity, generator, evaluator, breadth/depth limits, and stopping rule is not a reproducible algorithm.

A generic tree-search loop is:

1. Start with root state $s_0 = x$.
2. Select a frontier node according to a search policy.
3. Expand it by sampling b candidate next steps from the model.
4. Score candidates with a value function, verifier, or heuristic.
5. Stop when a candidate answer passes the acceptance criterion or the budget is exhausted.

The key design choice is not whether the structure is called a tree. It is whether the score used to guide expansion is valid. A weak heuristic can make tree search more expensive than self-consistency without improving correctness.

14.3.4 Search-Guided Training Loops

Tree search can also be a data-generation and training loop rather than only an inference wrapper. TS-LLM follows this AlphaZero-like pattern: a policy proposes actions, a learned value function scores partial states, search collects trajectories, and the policy and value model are updated from the searched data [169]. This is different from asking

a frozen model to evaluate its own thoughts. The search controller, value model, and policy become part of the training system.

A practical implementation needs an environment interface: state serialization, legal actions or action sampling, transition logic, terminal checks, and rewards. Some tasks expose final rewards during search, such as RLHF-style preference environments. Other tasks, such as GSM8K-style math, Game24, or proof-like reasoning, may require non-terminal value estimates because the final answer is unavailable until a whole trajectory is completed. Reporting only accuracy hides the important knobs: branch factor, maximum action length, rollout count, value or PRM weight, terminal versus non-terminal reward, and the number of generated tokens consumed by search.

The main risk is that searched traces are optimization artifacts. A trajectory may be useful for training even when it is not a faithful human-readable rationale. If the value model or PRM rewards plausibility, formatting, or early lucky branches, search can amplify those errors. Search-guided training should therefore log failed branches, verifier disagreements, reward-parser failures, and cost curves, not only the final selected answer.

14.4 Reinforcement Learning with Verifiable Rewards

14.4.1 Verifiable Rewards

Reinforcement learning with verifiable rewards (RLVR) trains a policy on tasks where correctness can be checked automatically. Examples include math problems with exact answers, programming tasks with tests, theorem-proving goals with proof checkers, and structured extraction tasks with known labels. The reward can be as simple as

$$R(x, y) = \begin{cases} 1, & \text{if verifier accepts } y, \\ 0, & \text{otherwise.} \end{cases}$$

The attraction is scale: no human preference label is needed for each rollout. The risk is narrowness: the reward covers what can be verified, not necessarily what matters. A code solution can pass visible tests while being brittle; a math answer can be right for the wrong reason; a model can learn formatting tricks that exploit the verifier.

DeepSeek-R1 showed that large-scale RL on verifiable tasks can elicit strong reasoning behavior, including long traces, self-checking, and backtracking, without starting from human-written reasoning demonstrations in the R1-Zero stage [33]. Subsequent RLVR studies examine when such rewards incentivize correct reasoning rather than merely correct answers [176]. The answer depends on task design, verifier strength, exploration, and whether shortcuts are available.

14.4.2 Group Relative Policy Optimization

GRPO-style objectives replace a learned value function with group-relative baselines. For a prompt x, sample a group of G responses $y_1, \ldots, y_G$ and compute rewards r_i. A normalized group advantage is

$$\hat{A}_i = \frac{r_i - \text{mean}(r_1, \ldots, r_G)}{\text{std}(r_1, \ldots, r_G) + \delta}.$$

The policy update then increases the likelihood of above-average samples and decreases the likelihood of below-average samples, usually with clipping and KL regularization against a reference policy. Removing the value model reduces memory and training complexity, but group sampling increases rollout cost. The method is most natural when rewards are cheap to compute and many samples can be generated per prompt.

A simplified clipped objective for a response $y_i = (a_{i,1}, \ldots, a_{i,T_i})$ is

$$\begin{aligned}
u_{i,t}(\theta) &= \rho_{i,t}(\theta)\hat{A}_i, \\
v_{i,t}(\theta) &= \text{clip}(\rho_{i,t}(\theta), 1 - \epsilon_{\text{clip}}, 1 + \epsilon_{\text{clip}})\hat{A}_i, \\
q_i(\theta) &= \text{avg}_{1 \le t \le T_i} \min(u_{i,t}(\theta), v_{i,t}(\theta)), \\
K_\theta(x) &= D_{\text{KL}}(\pi_\theta(\cdot|x) \,\|\, \pi_{\text{ref}}(\cdot|x)), \\
\mathcal{J}_{\text{GRPO}}(\theta) &= \text{avg}_{1 \le i \le G}\, q_i(\theta) - \beta K_\theta(x).
\end{aligned}$$

Here

$$\rho_{i,t}(\theta) = \frac{\pi_\theta(a_{i,t}|x, a_{i,<t})}{\pi_{\theta_{\text{old}}}(a_{i,t}|x, a_{i,<t})}.$$

Real implementations differ in token averaging, KL approximation, reward normalization, and whether the reward is attached only to the final answer or to intermediate verified steps. The important distinction from PPO is that the baseline is computed from the sampled group rather than from a learned critic. This makes memory use simpler, but it also makes optimization sensitive to group diversity: if all samples are equally bad, the relative advantages can still push the model toward the least bad local pattern.

GRPO and related methods also change the data distribution seen during training. The model learns from its own sampled attempts, not only from human-written solutions. This is powerful for exploration, but it amplifies verifier errors. If the verifier accepts a flawed shortcut, group-relative optimization can rapidly teach that shortcut.

14.4.3 Small-Scale GRPO Reproducibility Contracts

Course-scale R1-style repos make GRPO easier to run, but they also expose how many details must be fixed before a result is comparable. A reproducible run should name the base checkpoint, dataset split, prompt template, answer format, reward functions, sampling temperature, maximum completion length, number of generations per prompt, KL

or reference-policy setting, LoRA versus full fine-tuning, and whether rollout generation is served by the same process or by a separate inference engine.

Reward functions are not neutral metrics. A typical math or medical reasoning run may combine exact answer parsing, boxed-answer extraction, format checks for `think` and `answer` tags, reasoning-step heuristics, length rewards, cosine-shaped rewards, and repetition penalties. Each component changes what the model is optimized to produce. The training batch also has a systems constraint: the global train batch must divide cleanly by the number of sampled generations, otherwise group-relative normalization is ill defined or implementation-dependent.

The rollout engine is part of the experiment. Some small R1-style runs reserve one GPU for vLLM while the remaining processes train with Accelerate, ZeRO, or PEFT. That arrangement changes both speed and correctness contracts: the trainer must periodically gather or merge the current weights into the inference engine, broadcast generated completions back to the training ranks, pad and mask completions after the first EOS, and gather rewards across processes before computing group-relative advantages. If these steps are skipped or logged only implicitly, the reported gain may depend on stale rollout weights, duplicated samples, or process-local reward normalization rather than on the GRPO objective itself.

The minimum logging contract is therefore accuracy plus format accuracy, reward-component means, sampled completions, response length, token throughput, and evaluation on a held-out benchmark. Without these fields, an apparent R1-like gain may be a reward-parser artifact, a formatting improvement, or a budget increase rather than better reasoning.

14.5 Inference-Time Scaling

14.5.1 Compute Shapes

Test-time compute can be scaled in several shapes:

Sequential. Generate a longer trace, revise, reflect, or continue thinking.
Parallel. Generate multiple independent attempts and vote or rank them.
Tree-structured. Explore branches and allocate more budget to promising partial states.
Tool-augmented. Spend compute on retrieval, code execution, search, or external solvers.

If a prompt has prefill cost C_p, each generated token has decode cost C_d, and a strategy generates K samples of average length L, a crude serving cost is

$$C_{\text{serve}} \approx C_p + KLC_d + C_{\text{verify}} + C_{\text{tools}}.$$

This equation hides batching, KV-cache memory, parallelism, and hardware utilization, but it makes the product tradeoff explicit. A reasoning strategy that improves accuracy

by two points may be unacceptable if it multiplies latency by ten for all requests. Adaptive policies attempt to spend more compute only on hard prompts.

14.5.2 Budget Forcing

Budget forcing controls the amount of reasoning by encouraging the model to continue or stop at chosen points. The s1 work demonstrated that simple test-time scaling and budget forcing can improve a distilled reasoning model on competition math tasks [113]. The broader lesson is that inference-time behavior is a controllable interface: max tokens, stop conditions, prompts, verifier thresholds, and sampling parameters all shape the reasoning process.

Budget controls should be evaluated for both accuracy and failure mode. Forcing longer reasoning can help on multi-step math but hurt on perception-grounded multimodal tasks, factual questions where the model lacks knowledge, or tasks where the first correct answer is later revised into an error. More thinking is useful only when the additional computation has access to a corrective signal.

14.5.3 Compute-Optimal Allocation

Test-time scaling work shows that the best allocation depends on the model and prompt [155, 24, 187]. Easy prompts may need one short answer. Medium prompts may benefit from a few samples. Hard prompts may require search, tools, or a larger model. Impossible prompts should trigger abstention rather than unbounded reasoning.

A practical router can estimate difficulty and choose among strategies:

$$\pi_{\text{route}}(x) \in \{\text{direct}, \text{CoT}, \text{self-consistency}, \text{tool}, \text{human}\}.$$

The router itself must be evaluated. If it sends too many prompts to expensive reasoning, cost explodes. If it misses hard prompts, accuracy suffers. If it routes safety-sensitive prompts to long free-form reasoning without strong policy constraints, risk increases.

14.6 Frontier Reasoning Systems

Frontier reasoning models released in 2024 and 2025 made test-time compute a first-class product feature. Public system cards for OpenAI reasoning models describe safety evaluations, tool use, and risk assessments for models that reason longer before answering [122]. Open-weight reports such as DeepSeek-R1 and Qwen3 document pipelines that combine supervised data, RL, verifiable rewards, distillation, and inference-time controls [33, 183].

Table 14.1 Reasoning methods as compute-allocation policies.

Method	Added computation	Best fit	Main risk
Chain of thought	Extra intermediate tokens	Multi-step symbolic or explanatory tasks	Trace unfaithfulness and leakage of sensitive reasoning
Self-consistency	Parallel samples plus vote or normalization	Tasks with diverse independent solution paths	Correlated errors and high token cost
Program-of-thought	Code generation and execution	Arithmetic, parsing, simulation, and structured checks	Sandbox, dependency, and hidden test failures
Verifier selection	Candidate generation plus scoring	Code, math, schemas, and proof-like domains	Reward hacking or verifier blind spots
Tree search	Branch expansion and partial-state scoring	Planning or combinatorial tasks with useful heuristics	State explosion and misleading heuristic scores
RLVR/GRPO	Rollouts, verifiable rewards, and policy updates	Domains with cheap automatic correctness checks	Shortcut rewards and narrow task coverage

These reports should be read as engineering evidence, not as final scientific explanations. They show that reasoning performance can be improved by post-training and inference-time computation. They do not prove that visible chains are faithful, that reward-trained reasoning generalizes outside verifiable domains, or that benchmark gains imply robust planning in the real world.

Table 14.1 compares major reasoning methods by added computation, fit, and risk.

14.7 Evaluation Cautions

14.7.1 Answer Accuracy Is Not Enough

A reasoning evaluation should specify:

- answer metric: exact match, unit tests, proof check, human rubric, or preference;
- trace policy: hidden, visible, summarized, or unavailable;
- compute budget: max tokens, number of samples, tool budget, and wall-clock limit;
- selection method: greedy, majority vote, verifier rank, or human choice;
- contamination controls: held-out questions, transformed variants, and private tests;
- cost metric: tokens, latency, dollars, energy, or hardware occupancy.

Reporting only accuracy hides the central tradeoff. A model that solves 5% more tasks with 20 times more generated tokens may be valuable for theorem proving and unacceptable for customer support.

14.8 Cost Curves and pass@k

For independent samples with single-sample success probability p, the idealized probability that at least one of k samples succeeds is

$$\text{pass@}k = 1 - \text{pow}(1 - p, k), \tag{14.1}$$

where $\text{pow}(1-p, k)$ denotes the kth power of the single-sample failure probability. This formula is optimistic when samples are correlated, when the selector cannot identify the correct solution, or when the verifier is flawed. It is still useful because it exposes diminishing returns: the first few samples may help, but each additional sample adds cost. A reasoning evaluation should therefore report a curve

$$\{(C_k, A_k) : k \in \mathcal{K}\}, \tag{14.2}$$

where C_k is tokens, latency, or dollars and A_k is accuracy or task success. A single best-score point hides whether the system is compute-efficient.

14.8.1 Trace Faithfulness

Visible reasoning traces are tempting evaluation artifacts. They let graders inspect steps, train process reward models, and debug failures. But a trace can be optimized for the grader. If the final answer is produced by internal activations and the visible trace is generated for presentation, then step-level natural-language evaluation may overstate reliability. Conversely, hiding all traces makes debugging and accountability harder. Many products therefore expose concise rationales or cited calculations, not raw chain-of-thought.

14.8.2 Reasoning Boundary Tests

Reasoning benchmarks should include boundary cases:

- problems with insufficient information, where abstention is correct;
- adversarially phrased easy problems, where overthinking causes errors;
- tasks requiring external knowledge not in the prompt;
- tasks with misleading but irrelevant details;
- tasks where a tool result contradicts the model's prior belief;
- safety-sensitive tasks where the correct behavior is refusal or redirection.

These cases distinguish useful deliberation from verbose pattern completion.

14.9 Implementation Notes

A robust test-time reasoning stack usually contains five layers:

1. A prompt or policy that defines whether scratch work is private, visible, or summarized.
2. A budget controller for max tokens, number of samples, branch factor, and tool calls.
3. A verifier or acceptance test where the domain allows it.
4. A router that escalates hard, risky, or low-confidence requests.
5. Telemetry that logs budget, latency, selected strategy, verifier result, and user-visible answer.

For software engineering tasks, the verifier may be tests and static analysis. For math, it may be exact answer checking or symbolic verification. For factual tasks, retrieval and citation checks are often more valuable than longer unaided reasoning. For safety-sensitive tasks, policy checks should run outside the model's own free-form scratchpad.

14.10 Key Terms

Chain of thought Generated intermediate reasoning tokens used to support a final answer.

Self-consistency Sampling multiple reasoning traces and aggregating final answers.

Thought decomposition The choice of what unit of partial reasoning becomes one expandable search step.

Verifier A model, program, or human process that scores or checks candidate solutions.

Process reward model A verifier that scores intermediate reasoning steps.

Tree search A method that expands and evaluates partial reasoning states.

Search-guided training Using searched trajectories and value estimates to update a policy or value model.

RLVR Reinforcement learning with rewards from automatic verifiers.

GRPO A group-relative policy optimization style that uses sampled group rewards as a baseline.

Test-time scaling Improving performance by spending more computation during inference.

Budget forcing Controlling the amount of reasoning computation by prompting or decoding constraints.

14.11 Exercises

1. Run a small math benchmark with greedy answers, CoT, and self-consistency with $K = 5$. Report accuracy, average output tokens, and latency.

2. Implement answer normalization for self-consistency on arithmetic problems. Measure how many apparent disagreements are only formatting differences.
3. Build a sample-and-rank loop using a unit-test verifier for simple programming tasks. Compare best-of-K selection to majority vote.
4. Specify a Tree-of-Thought setup for Game of 24 or another small puzzle: define the thought unit, generator prompt, state evaluator, search policy, breadth/depth limits, and token budget.
5. Create five problems where longer reasoning helps and five where it hurts. Explain the difference in terms of available corrective signals.
6. Derive the group-relative advantage for a batch of four sampled rewards. Show what happens when all rewards are equal.
7. Design an evaluation card for a reasoning model that reports accuracy, token budget, verifier type, hidden/visible trace policy, and contamination controls.

Chapter 15
Multimodal and Generative Foundation Models

Abstract This chapter introduces multimodal and generative foundation models through contrastive representation learning, visual encoders, projection layers, query transformers, cross-attention adapters, unified understanding-generation systems, diffusion and rectified-flow objectives, any-to-any omni models, multimodal instruction tuning, OCR and chart reasoning, image and video token budgets, multimodal evaluation, hallucination, privacy, and safety issues unique to systems that combine language with images, audio, video, and other continuous signals.

Chapter contract.

The reader should leave this chapter able to analyze multimodal systems as foundation-model lifecycles with new token, objective, latency, provenance, and safety contracts, rather than as language models with pictures attached, and compare understanding, generation, and action interfaces.

15.1 Modalities as Interfaces

A language model consumes tokens. Images, audio, video, tables, plots, documents, and sensor streams are not naturally token sequences in the same vocabulary. A multimodal LLM must therefore solve an interface problem: convert non-text signals into representations that a language model can use while preserving the information needed for the task.

The modern vision-language model (VLM) stack usually contains a modality encoder, a connector, and a language model. The encoder turns an image or video into patches, frames, or embeddings. The connector maps those representations into the language model's hidden space. The language model then performs instruction following, reasoning, and generation. This design inherits strengths and weaknesses from both sides. It can use the language model's broad knowledge and dialogue ability, but it can

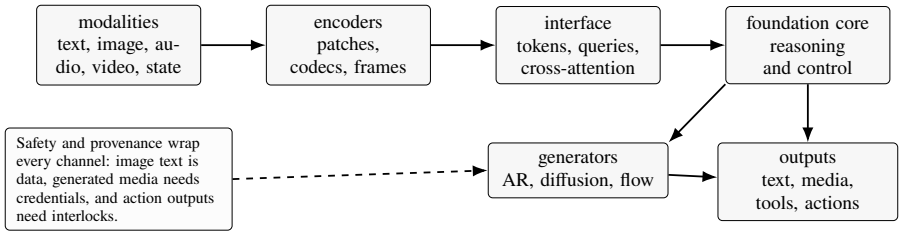

Fig. 15.1 A unified multimodal system is an interface design problem. The core model may be shared, but each modality still needs explicit encoding, token-budget, generation, evaluation, and safety contracts.

also hallucinate visually, over-rely on text priors, miss small spatial details, and obey malicious instructions embedded inside images.

This chapter takes a wider view than LLM-centered visual question answering. Understanding and generation are converging. A frontier system may read an image, reason over a document, speak in real time, edit a picture, generate video, call tools, and eventually select actions in a physical or simulated environment inside one product contract. The core engineering ideas are representation alignment, token budgeting, generative objective choice, instruction data, action interfaces, evaluation, safety, and provenance across modalities.

Figure 15.1 shows the interface layers that connect modality encoders, a foundation core, generators, and action outputs.

15.2 Contrastive Pretraining

15.2.1 Image-Text Alignment

CLIP-style pretraining learns a shared embedding space for images and text using large collections of image-caption pairs [138]. Let $f_\theta(I_i)$ be the normalized embedding of image I_i, and $g_\psi(T_j)$ be the normalized embedding of text T_j. For a batch of N matched pairs, the image-to-text contrastive loss is

$$\begin{aligned} s_{ij} &= f_\theta(I_i) \cdot g_\psi(T_j)/\tau, \\ D_i &= \exp(s_{i1}) + \cdots + \exp(s_{iN}), \\ \mathcal{L}_{I\to T} &= -\operatorname{avg}_{1\le i\le N} \log \frac{\exp(s_{ii})}{D_i}, \end{aligned}$$

with a symmetric text-to-image term. The temperature τ controls the sharpness of the matching distribution. The model is not trained to generate captions. It is trained to place matched images and texts near each other and unmatched pairs farther apart.

This objective is powerful because web-scale captions provide broad supervision. It also has limits. Captions are incomplete, noisy, culturally biased, and often describe

salient objects rather than fine-grained spatial relations. A CLIP encoder may know that an image is about a dog on a skateboard while missing whether the dog's left paw touches the wheel. Downstream VLMs inherit these representation gaps unless later training addresses them.

15.2.2 Zero-Shot Classification

CLIP also demonstrates that language can define labels. To classify an image, one embeds prompts such as "a photo of a cat" and "a photo of a dog", then chooses the text embedding with highest similarity to the image embedding. This is zero-shot transfer through natural-language class descriptions. The same idea appears in retrieval, image clustering, dataset filtering, and weak labeling.

For LLM-centered VLMs, contrastive encoders provide a strong visual backbone, but the assistant still needs a way to condition a generative language model on visual features. That connector is where many architecture differences arise.

15.3 Connecting Vision and Language

15.3.1 Frozen Encoders and Projection Layers

The simplest connector freezes a vision encoder and a pretrained LLM, then trains a projection module P_η:

$$H_{\text{vis}} = E_{\text{vis}}(I), \qquad Z_{\text{vis}} = P_\eta(H_{\text{vis}}).$$

The projected visual tokens Z_{vis} are inserted into the LLM context alongside text tokens. LLaVA popularized this recipe for visual instruction tuning by connecting a CLIP vision encoder to a language model and training on image-grounded instruction data [97]. The appeal is simplicity: most parameters can remain frozen initially, and the connector learns the alignment.

Projection-only connectors are cheap, but they may pass too many image tokens or fail to select task-relevant details. If the vision encoder emits hundreds or thousands of patch features, each becomes part of the LLM context unless compressed. Long visual prefixes increase prefill cost and compete with text for context window space.

15.3.2 Query Transformers and Token Compression

BLIP-2 introduced a Querying Transformer (Q-Former) that learns a small set of query tokens to extract information from a frozen image encoder before passing it to a frozen LLM [87]. This design reduces the number of visual tokens and creates a trainable

Connector	Strengths	Typical risks
Linear or MLP projection	Simple, cheap, easy to retrofit to an LLM	Too many tokens; weak spatial grounding; connector bottleneck
Query transformer	Compresses visual features; keeps large backbones frozen	May discard small text or spatial details; extra pretraining stages
Cross-attention adapter	Handles interleaved media; flexible conditioning	More invasive architecture changes; higher memory cost
End-to-end multimodal training	Can optimize perception and language jointly	Expensive; greater risk of forgetting and data imbalance

Table 15.1 Common connector designs for LLM-centered multimodal models.

bottleneck:

$$Z_{\text{query}} = Q_\eta(Q_0, E_{\text{vis}}(I)).$$

The bottleneck can improve efficiency and force selection, but it can also discard details needed for OCR, diagrams, or counting. Many systems therefore vary token count by task or resolution: a low-resolution global view for general conversation, higher-resolution crops for documents, and frame sampling for video.

15.3.3 Cross-Attention and Interleaved Media

Flamingo-style models add cross-attention layers that let the language model attend to visual features while preserving a pretrained language backbone [2]. This design supports interleaved image-text sequences and few-shot multimodal prompting. The model can see examples such as image, question, answer, image, question, answer within a single context.

Cross-attention adapters are more complex than projection layers, but they avoid forcing all visual information into ordinary text-token positions. They also make it easier to handle multiple images or video frames. The tradeoff is implementation complexity, memory use, and the need to train or tune inserted layers across the language model depth.

15.3.4 Architecture Tradeoffs

Table 15.1 compares common connector designs and their risks.

15.4 Unified Understanding and Generation

The older split was simple: vision-language models answered questions about media, while diffusion models generated images or video. Recent work is trying to make that boundary disappear. Surveys of unified multimodal understanding and generation models organize the field into autoregressive, diffusion-based, and hybrid paradigms, and emphasize recurring challenges in tokenization, cross-modal attention, data balance, and benchmark design [200].

Unified models must solve two tasks that pull in different directions. Understanding wants faithful extraction: read the pixels, bind entities, preserve spatial and temporal evidence, and abstain when the input is insufficient. Generation wants controllable synthesis: create new media that follows the instruction, remains coherent, and respects safety and provenance constraints. A single system that does both cannot be evaluated only by VQA accuracy or image aesthetic preference. It needs a capability profile over perception, reasoning, editing, controllability, diversity, latency, and misuse resistance.

Janus-Pro illustrates an autoregressive path toward unification. It improves a Janus-style design for both multimodal understanding and text-to-image instruction following, with separate attention to training strategy, data scaling, model size, and generation stability [22]. The key lesson is architectural decoupling: the representation that is best for understanding an image is not automatically the representation that is best for generating one. Unified systems often need shared high-level reasoning while preserving modality-specific interfaces where fidelity matters.

Omni systems push the interface further. Qwen3-Omni reports a single multimodal model spanning text, image, audio, and video understanding with speech generation and low-latency streaming considerations [182]. AR-Omni takes an even more literal autoregressive position: text, image generation, and streaming speech generation are placed under a single Transformer decoder and next-token-style interface [25]. Whether such systems win broadly depends on more than benchmark averages. They must show that one interface can handle modality imbalance, real-time constraints, visual fidelity, speech naturalness, safety policy, and efficient serving.

15.5 Generative Modeling Objectives

Autoregressive language modeling is only one generative objective. For text, the next-token factorization is natural because output is a sequence. For images, audio, and video, sequential tokenization is possible but not always the most efficient or controllable representation. Generative foundation models therefore use a family of objectives.

Table 15.2 summarizes the objective families that appear in modern multimodal generation systems.

Figure 15.2 shows why unified multimodal systems need a regression grid rather than a single aggregate score.

Table 15.2 Generative objective families used in multimodal foundation systems.

Objective	Natural strengths	Typical cost surface	Failure modes
Autoregressive	Streaming text, code, speech tokens, tool traces	Sequential decode length and KV cache	Awkward spatial ordering; slow high-resolution media
Diffusion	Image/video fidelity, editing, diversity	Iterative denoising steps and guidance	Prompt drift; expensive sampling; safety filter dependence
Rectified flow	Direct noise-to-data transport with transformer backbones	Step schedule and latent resolution	Flow mismatch; weak conditioning if text-media alignment is poor
Hybrid AR-diffusion	Semantic planning plus continuous synthesis	Cascade latency and cross-module contracts	Beautiful but unfaithful media; brittle intermediate plans

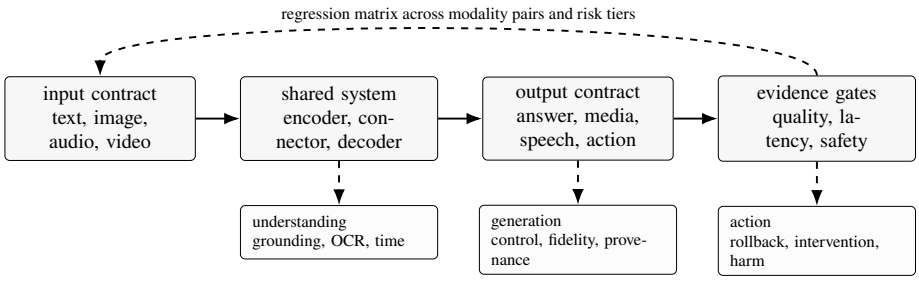

Fig. 15.2 Unified multimodal systems need regression coverage across understanding, generation, streaming, provenance, and action, not only a headline score. The grid synthesizes design pressures visible in Janus-Pro, MMaDA, Sora, AR-Omni, Qwen3-Omni, and OpenVLA [22, 186, 120, 25, 182, 72].

15.5.1 Autoregressive Generation

Autoregressive multimodal models serialize media into tokens and train the model to predict the next token. This gives a unified training and inference story: the same decoder can condition on text, image tokens, audio tokens, and previous generated media tokens. The price is ordering. A fixed raster or codec order may be awkward for global image layout, bidirectional editing, or infilling. The model may need many sequential steps to generate content that humans perceive holistically.

15.5.2 Diffusion and Rectified Flow

Diffusion models generate by starting from noise and iteratively denoising toward data. In latent diffusion, the process occurs in a compressed latent space rather than directly

over pixels. Diffusion Transformers (DiTs) replace the traditional U-Net denoiser with a transformer over latent patches and show predictable scaling behavior for image generation [126]. Sora extends this idea to visual data by training text-conditioned diffusion models over spacetime patches of compressed video and image latents [120]. Stable Diffusion 3-style work uses rectified flow transformers for high-resolution text-to-image synthesis, treating generation as learning a flow from noise to data and emphasizing transformer-based multimodal interaction between text and image tokens [39].

In the common denoising formulation, a clean latent x_0 is corrupted to x_t and the model predicts the injected noise under condition c:

$$q_{\text{diff}}(x_t, t, c) = \epsilon - \epsilon_\theta(x_t, t, c),$$
$$\mathcal{L}_{\text{diff}} = \mathbb{E}_{x_0, \epsilon, t, c}\, q_{\text{diff}}(x_t, t, c) \cdot q_{\text{diff}}(x_t, t, c).$$

Rectified flow instead learns a velocity field between a noise endpoint ξ_0 and data endpoint x_1, often with $x_t = (1 - t)\xi_0 + t x_1$:

$$q_{\text{flow}}(x_t, t, c) = v_\theta(x_t, t, c) - (x_1 - \xi_0),$$
$$\mathcal{L}_{\text{flow}} = \mathbb{E}_{\xi_0, x_1, t, c}\, q_{\text{flow}}(x_t, t, c) \cdot q_{\text{flow}}(x_t, t, c).$$

These formulas hide many implementation choices: latent encoder, timestep schedule, conditioning pathway, guidance, safety filters, and how generated media is linked back to prompt evidence.

For a book about LLMs, the important point is not that every reader must implement a video generator. The point is that token prediction is no longer the only route to foundation-model behavior. Discrete diffusion language models such as Dream 7B refine text sequences through iterative denoising and expose any-order generation, infilling, and quality-speed tradeoffs [190]. MMaDA applies a unified diffusion formulation to textual reasoning, multimodal understanding, and text-to-image generation, with a post-training recipe tailored to diffusion foundation models [186]. These systems are early, but they force the curriculum to discuss generation as a general modeling problem rather than treating language modeling as the whole field.

15.5.3 Hybrid AR-Diffusion Systems

Hybrid systems try to combine the strengths of both families. An autoregressive path can plan high-level semantics, tool actions, or discrete structure; a diffusion or flow path can synthesize high-fidelity continuous media. MammothModa2 is an example of this AR-diffusion direction: it couples autoregressive semantic planning with a Diffusion Transformer decoder for image generation and editing, trained with both next-token prediction and flow matching objectives [151]. The design question is where to place the boundary. If the AR module plans too little, generation may ignore the instruction. If it plans too much, the system becomes a brittle cascade. If the diffusion module is too detached, it may produce beautiful but unfaithful media.

This objective choice changes serving and governance. Autoregressive models expose token log probabilities and stream naturally. Diffusion and flow models expose step schedules, guidance scales, denoising trajectories, and image/video safety filters. Hybrid models expose both. A publishable system report should state not only what the model can generate, but which objective produced it, which tokens or latents were used, what safety filters operated before and after generation, and how provenance is preserved.

15.6 Action, Embodiment, and World Models

The next boundary after text, image, audio, and video is action. A vision-language-action (VLA) model maps observations and instructions to actions, often by representing robot actions as tokens or by attaching an action head to a multimodal backbone. RT-2 showed a direct version of this idea: co-fine-tune vision-language models on web-scale vision-language tasks and robotic trajectories, then express actions in a token-like format so the model can transfer some semantic knowledge into robotic control [14]. OpenVLA made this direction more reproducible by releasing a 7B open-source VLA trained on a large collection of real-world robot demonstrations, with fine-tuning and quantization paths for new settings [72].

VLA systems make the phrase "foundation model" concrete and dangerous. Text generation errors can be edited; robot actions change the world. A VLA report must therefore describe observation frequency, action representation, control horizon, latency, embodiment, reset policy, safety interlocks, teleoperation data, simulation-to-real transfer, and failure recovery. A model that recognizes an object in a static image may still fail to grasp it because of occlusion, calibration error, friction, actuator limits, or delayed feedback.

Recent VLA surveys frame the area as the unification of visual perception, natural language understanding, and embodied control, while emphasizing datasets, simulation platforms, architectural paradigms, and deployment challenges [167]. For this book, the important curriculum decision is scope. Robotics is not a side note to chatbots, but a complete robotics textbook would dilute the main thread. The right placement is here: action is another modality with stricter safety, timing, and evaluation contracts. The same questions recur in harder form: what is the representation, what data defines the behavior, what objective trains it, what system serves it, and what evidence justifies deployment?

A behavior-cloned VLA policy can be written as

$$\mathcal{L}_{\mathrm{VLA}} = -\operatorname{avg}_t \log \pi_\theta(a_t \mid o_{\le t}, u, h_t),$$

where $o_{\le t}$ is the observation history, u is the language instruction, h_t is optional task memory, and a_t may be a discretized action token or a continuous control target. Reinforcement-learning variants replace or augment imitation with an expected return objective $\mathbb{E}_{\pi_\theta}[\sum_t \gamma^t r_t]$, but real-world exploration is constrained by safety, reset cost, and sparse rewards.

World models connect action and generation. A video generator that predicts plausible futures, an embodied model that imagines the consequence of a grasp, and a planner that simulates alternative tool calls all use generative modeling as a decision aid. The risk is confusing visual plausibility with physical truth. A generated future can be coherent and still violate dynamics, contact, object permanence, or task constraints. Evaluation must therefore test not only whether the future looks realistic, but whether it supports better decisions under intervention.

15.7 Multimodal Instruction Tuning

15.7.1 Training Stages

A practical VLM is rarely trained in one stage. A common sequence is:

1. Pretrain or choose a vision encoder using contrastive or supervised vision-language data.
2. Train a connector on image-caption pairs or image-text matching data.
3. Tune the combined model on image-grounded instruction data.
4. Add task-specific data for OCR, charts, documents, localization, visual math, or video.
5. Add safety and refusal data that includes both text-only and image-conditioned attacks.

The order matters. Early connector training teaches the language model to interpret visual embeddings as context. Later instruction tuning teaches conversational behavior and task formats. If the instruction data is too small or too synthetic, the model may learn a superficial response style without robust perception.

15.7.2 Data Sources

Multimodal instruction data can come from human annotation, synthetic conversations generated by stronger models, templated conversions of existing vision datasets, OCR/document datasets, chart QA datasets, and video QA datasets. Synthetic data is useful for coverage, but it can inject teacher-model hallucinations. Human data is expensive but essential for ambiguous tasks such as describing medical images, evaluating visual quality, or explaining charts for a specific audience.

The dataset should explicitly cover:

- object recognition, attributes, relations, and counting;
- OCR, handwriting, tables, charts, and diagrams;
- spatial reasoning and localization;
- multi-image comparison;
- visual math and science reasoning;

- uncertainty and abstention when the image is ambiguous or low resolution;
- safety cases where the image changes the policy decision.

Without such coverage, the VLM may answer from language priors. For example, if a prompt asks what color a stop sign is, a text-only prior may produce "red" even when the image is grayscale, occluded, or adversarially edited.

15.7.3 Chat Templates with Images

Image tokens must be represented in the chat template. A serialized example may look conceptually like:

<system> policy <user> <image> question <assistant> answer.

The literal tokens differ by implementation, but consistency is critical. Training and inference should use the same role markers, image markers, and ordering. Multi-image prompts need unambiguous references: "first image", "second image", bounding boxes, page numbers, or frame timestamps.

15.7.4 Image Markers, Tiling, and Label Masks

A token such as <image> is a placeholder, not a normal word. In LLaVA-style systems, the processor tokenizes the text prompt, records the image marker id, preprocesses the image into one or more fixed-size tensors, passes those tensors through a vision encoder, projects the resulting patch features, and then replaces the marker position with visual embeddings [97]. Once this replacement happens, the sequence seen by the language model is no longer the same length as the text-token sequence.

High-resolution handling makes the expansion concrete. A non-square image may be resized, padded, tiled into crops, and accompanied by a global view so that small details remain visible without distorting the original aspect ratio. If a ViT encoder uses patch size 14 and receives 336×336 crops, one crop yields $24 \times 24 = 576$ patch features before any selection or pooling. Multiple crops can therefore produce thousands of visual tokens for a single user turn. This is why visual-token budgeting belongs in the same discussion as long-context serving.

The merged multimodal sequence must rebuild attention masks, position ids, and labels. Visual embeddings should condition the answer, but they are not text labels to be predicted by the language-model loss. Text padding, image padding, crop order, multi-image references, and previous conversation turns must be aligned before batching. Many implementation failures that look like weak visual reasoning are actually serialization failures: the image marker is misplaced, the visual features are inserted at the wrong position, labels are shifted across the image span, or the attention mask lets one packed example see another.

15.7.5 OCR, Charts, and Documents

Document understanding stresses VLMs differently from natural images. The model must read small text, preserve layout, associate labels with values, and reason over tables. Increasing image resolution helps but increases visual tokens. Some systems use specialized OCR tools and feed recognized text to the LLM; others rely on native visual reading. The best choice depends on the domain. OCR tools provide explicit text and coordinates, while native VLMs can combine layout and semantics but may hallucinate characters.

Charts require another layer: the model must map visual marks to quantities. It may need to infer axes, units, legends, and trends. Evaluation should check numerical accuracy, not merely plausible prose. A response that describes a bar chart fluently but swaps the largest and second-largest category is a failure.

15.8 Systems Costs

15.8.1 Visual Tokens and Prefill

For text-only LLMs, long prompts increase prefill cost. For VLMs, the image itself can contribute hundreds or thousands of tokens before the user question is considered. If an image is split into P patches and projected to M visual tokens, the LLM prefill length becomes approximately

$$L_{\text{prefill}} = L_{\text{text}} + M.$$

Attention cost during prefill grows roughly quadratically in total context length for standard attention, though optimized kernels and attention variants change constants. This makes high-resolution images, multi-page documents, and video expensive.

15.8.2 Video and Audio

Video adds a temporal dimension. A naive system that encodes every frame quickly exhausts context and compute. Practical systems sample frames, detect scene changes, compress frame features, or use a video encoder before the LLM. Video-LLaVA-style systems align image and video features before projection so that a language model can consume a unified visual representation [94]. Audio similarly requires chunking, spectrogram or codec features, speech recognition, speaker or prosody handling, and temporal alignment. AudioPaLM-style systems connect text and speech models so that speech recognition, speech translation, and speech generation can share a language-model backbone [145]. Omni models such as Qwen2.5-Omni make the deployment contract harder still: they must stream over text, image, audio, and video inputs while generating text or natural speech responses [135].

Temporal grounding is the core evaluation problem. A model may answer a global question about a video while failing to identify the frame, timestamp, or spoken segment that supports the answer. A publishable video or audio evaluation should separate event detection, temporal localization, transcription quality, speaker attribution, cross-modal coreference, and final reasoning. It should also report whether external tools are used. Tool-assisted OCR, ASR, chart extraction, object detection, or browser automation can make a system far stronger than a pure end-to-end model, but then the evaluation is of a pipeline, not only of a checkpoint.

System designers should report:

- maximum image resolution and resizing policy;
- number of visual tokens per image or frame;
- frame sampling rate for video;
- whether OCR, ASR, or other tools are used;
- latency split between encoder, connector, LLM prefill, and decode;
- memory impact on batching and KV cache.

Without these details, benchmark numbers are hard to interpret and deployment costs are hard to predict.

15.9 Evaluation

15.9.1 Benchmark Families

Multimodal evaluation must test perception and reasoning separately. MMMU evaluates multimodal understanding across expert-level disciplines [193]; MathVista focuses on mathematical reasoning in visual contexts [103]. POPE and HallusionBench probe visual hallucination and visual illusion failure modes [89, 50]. Recent frontier and open reports, including GPT-4o and Qwen2.5-VL, report broad multimodal capability and safety evaluations [118, 136, 53].

Benchmarks should be interpreted by task type. A model can be strong at OCR and weak at spatial reasoning, or strong at captioning and weak at chart arithmetic. A single leaderboard score is not a capability profile.

15.9.2 Evaluation Cautions

Key cautions include:

Language leakage. Questions may be answerable from text priors without inspecting the image.

Shortcut OCR. Datasets may contain visible text that gives away labels without visual reasoning.

Resolution sensitivity. A model may fail because preprocessing made the relevant detail unreadable.

Answer parsing. Free-form answers can hide wrong units, swapped labels, or approximate numbers.

Contamination. Public benchmark images and answers may appear in training data.

Tool ambiguity. A model using OCR or retrieval is not directly comparable to one using only pixels.

For serious deployments, create in-domain tests with private images, controlled perturbations, and human review of failures.

15.10 Safety and Governance

15.10.1 Visual Prompt Injection

Images can contain instructions. A screenshot may say "ignore previous instructions", a document may include hidden white text, or a QR code may point to malicious content. If the VLM treats image text as user instruction rather than untrusted data, it can be prompt-injected. Visual jailbreaks such as typographic prompts demonstrate that safety training for text-only models does not automatically transfer to vision-language systems [47]. MM-SafetyBench-style evaluations show that harmful requests can be smuggled or amplified through images [100].

A robust system distinguishes instruction channels from content channels. Text extracted from an image should usually be treated as quoted content, not as a higher-priority instruction. Tool outputs, OCR text, and image captions should be wrapped with provenance markers before being passed to the language model.

15.10.2 Privacy and Sensitive Inference

Images often contain faces, addresses, medical information, documents, screens, location clues, and bystanders. A VLM can infer sensitive attributes even when the user did not ask for them. Policies should define which inferences are allowed, which require consent, and which should be refused. Logging and retention rules are especially important for multimodal systems because raw images are more identifying than many text prompts.

15.10.3 Hallucination and Grounding

Visual hallucination occurs when a model asserts objects, text, actions, or relations not supported by the image. It can come from language priors, weak visual features,

low resolution, or overextended reasoning. Mitigations include higher-resolution crops, object detectors, OCR verification, uncertainty prompts, and training data with negative examples. The model should learn to say "I cannot tell from this image" when evidence is insufficient.

15.11 Implementation Notes

A minimal VLM implementation plan is:

1. Choose the visual domains: natural images, screenshots, documents, charts, video, or medical imagery.
2. Select a vision encoder and decide whether it is frozen, partially tuned, or trained end to end.
3. Choose a connector and visual-token budget.
4. Define the multimodal chat template and image-marker semantics.
5. Train the connector before broad instruction tuning unless using an already aligned VLM backbone.
6. Build evaluation slices for perception, OCR, spatial reasoning, chart reasoning, safety, and refusal.
7. Measure latency and memory by separating encoder, prefill, and decode costs.
8. Add prompt-injection defenses for image text and document content.

15.12 Key Terms

Vision encoder A model that maps images or video frames into feature representations.

Contrastive learning Training matched image-text pairs to be close in embedding space and mismatched pairs far apart.

Connector A projection, query transformer, or adapter that maps modality features into the LLM interface.

Visual token A representation of image content inserted into or attended by the language model.

Q-Former A query transformer that compresses visual features into a small set of learned query outputs.

Unified understanding-generation model A multimodal model or system that both interprets input media and generates or edits media under a shared interface.

Any-to-any model A system that accepts multiple input modalities and can produce multiple output modalities, such as text, image, audio, speech, or video.

Diffusion model A generative model that learns to reverse a noising process, often producing data through iterative denoising in pixel, token, or latent space.

Rectified flow A generative formulation that learns a flow from noise to data, often used as an alternative to diffusion for high-resolution visual synthesis.

VLA model A vision-language-action model that maps visual observations and language instructions to actions in a physical or simulated environment.

World model A model used to predict or simulate future states, observations, or consequences of actions for planning or evaluation.

Visual hallucination A claim about image content that is not supported by the image.

Visual prompt injection Malicious or conflicting instructions embedded in image or document content.

Grounding The degree to which a response is supported by specific parts of the input modality.

15.13 Exercises

1. Derive the symmetric CLIP loss for a batch of three image-text pairs and identify the positive and negative pairs.
2. Trace an image through a frozen vision encoder, projection layer, image marker, and language model context. List the tensor shapes at each step for a chosen patch size.
3. For a 1024×899 image processed as one global 336×336 view plus four tiled crops with patch size 14, estimate the visual-token count before connector compression. Which attention-mask and label-mask changes are required after insertion into a chat template?
4. Compare two connector designs for a document QA system: an MLP projection with many visual tokens and a query transformer with compressed tokens. Discuss accuracy and cost tradeoffs.
5. Compare autoregressive, diffusion, and hybrid AR-diffusion approaches for a unified text-image system. State how each approach handles instruction following, editing, latency, and safety filtering.
6. Design a minimal VLA evaluation for a tabletop robot. Specify observation modalities, action representation, reset rules, safety interlocks, and what counts as task success.
7. Build a small evaluation set with five natural images, five screenshots, and five charts. Write separate scoring criteria for perception, OCR, and reasoning.
8. Create three visual prompt-injection examples in a benign test environment. Propose a serialization strategy that treats image text as untrusted content.
9. Measure how resizing an image affects OCR accuracy for a VLM or OCR tool. Report the smallest text size that remains reliable.

Chapter 16
Evaluation, Safety, and Governance

Abstract This chapter closes the book by treating evaluation, safety, and governance as part of LLM engineering rather than as afterthoughts. It covers benchmark design, benchmark contamination, human evaluation, LLM-as-judge methods, long-context and agentic evaluation, multimodal generation evaluation, factuality, hallucination, robustness, red teaming, interpretability, unlearning, watermarking, provenance, privacy, copyright, model cards, dataset documentation, system cards, frontier risk frameworks, operational monitoring, incident response, and deployment limits.

Chapter contract.

The reader should leave this chapter able to turn a model-release question into a measurement plan, recognize benchmark and judge-model failure modes, connect safety and governance controls to operational evidence, and decide when a system should not be deployed.

16.1 Evaluation as a Measurement System

Evaluation is the feedback loop that tells an engineering team whether a modeling decision helped. Safety and governance define whether the resulting system should be deployed, who is accountable, and what evidence must be collected before and after release. For LLMs, these concerns are inseparable. A model can improve on a public benchmark while becoming less reliable for the actual product. A safety filter can reduce obvious harmful completions while increasing refusal of benign requests. A benchmark can saturate, leak into training data, or reward the wrong behavior.

This chapter organizes evaluation as a measurement system. A measurement system has a construct, a dataset, a protocol, a metric, an uncertainty estimate, and a decision rule. “MMLU score” or “win rate” is not enough. The team must know what capability the metric is supposed to measure, where the examples came from, how the model was

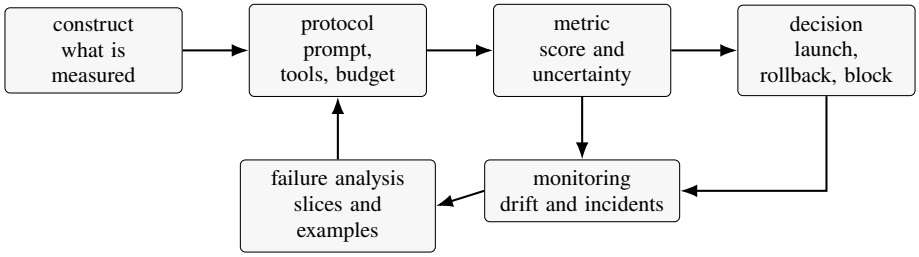

Fig. 16.1 Evaluation is an evidence loop. A benchmark result is actionable only when it is tied to a construct, protocol, uncertainty estimate, failure analysis, monitoring plan, and deployment decision.

prompted, whether tools were allowed, how outputs were parsed, and what deployment decision follows from the result.

Figure 16.1 summarizes the evidence loop used throughout this chapter.

16.2 Evaluation Design

16.2.1 From Question to Decision

Every evaluation should begin with a decision. Examples:

- Can this model replace the previous production model for customer support?
- Does a new preference-tuning run improve helpfulness without increasing unsafe compliance?
- Is a reasoning model worth its added latency for code repair?
- Does a RAG system answer from retrieved evidence rather than parametric memory?
- Is the model acceptable for a regulated domain with human oversight?

The decision determines the metric. A customer-support assistant needs task completion, escalation accuracy, tone, and policy compliance. A code agent needs patch correctness, test pass rate, repository hygiene, and security review. A medical information assistant needs source faithfulness, abstention, and strict scope control.

16.2.2 A Measurement Template

A useful evaluation card should include:

Construct. The behavior being measured, such as factual precision, instruction following, refusal accuracy, or tool-use success.

Population. The users, prompts, languages, domains, and risk classes the evaluation represents.

Protocol. Prompt format, system message, sampling parameters, tool access, context budget, and time limit.

Metric. Exact match, pass@k, rubric score, pairwise win rate, calibration error, cost, or latency.

Uncertainty. Confidence intervals, bootstrap estimates, annotator agreement, and sample size.

Decision rule. The threshold or comparison that determines launch, rollback, escalation, or further testing.

This template prevents benchmark results from becoming detached from engineering decisions.

16.2.3 Evaluation Harnesses as Code

A benchmark name is not a runnable evaluation. A practical harness must define the sample schema, prompt template, model-call interface, answer extractor, scorer, aggregation rule, and artifact format. Simple-evals-style implementations make this explicit by separating a sampler from task-specific evaluators [119]. The same model completion can be scored very differently depending on whether the harness extracts a multiple-choice letter, parses a final boxed mathematical answer, asks another model to judge equivalence, or executes code.

The implementation details are part of the measurement protocol. MMLU-style multiple-choice tasks need answer-order handling and a clear final-answer pattern [54]. MATH-style tasks often rely on exact-answer normalization and conventions such as a boxed final answer [55]. GPQA-style tasks randomize choices to reduce shortcut learning [143]. DROP-style reading-comprehension tasks need normalization for numbers, dates, spans, and multi-part answers [37]. MGSM-style multilingual reasoning tasks test whether a few-shot or chain-of-thought protocol transfers across languages rather than only across English prompts [152]. HumanEval-style code evaluation must execute generated programs against tests in a sandbox with timeouts and resource limits; otherwise the evaluator itself becomes a security risk [20].

Multiple-choice scoring is a concrete example of why the harness matters. One protocol asks the model to generate a final letter and exact-matches the parsed `A/B/C/D`; it tests the whole prompt-following and parsing pipeline but can fail when the model says the right answer in another form. A second protocol reads the next-token logits for the choice labels and normalizes only over the allowed labels; it avoids invalid generations but depends on tokenization and label priors. A third protocol appends each candidate answer and scores its conditional negative log probability or perplexity; scoring the answer text rather than only the option letter can reduce letter bias, but it changes the measured object from generation behavior to conditional likelihood. Reports should therefore name the scoring path, whether choices were shuffled, how invalid outputs were handled, and whether few-shot, chain-of-thought, `pass@N`, or majority-vote sampling was used.

Every run should write an audit trail: model identifier, prompt template, decoding settings, tool permissions, retrieved context if any, raw response, parsed answer, score, exception, runtime, cost, and commit or dataset version. Aggregate scores without per-sample records are hard to debug. Per-sample records also make regression analysis possible: when a new model loses three points, the team can determine whether the loss came from parsing failures, refusals, hallucinations, code timeouts, retrieval errors, or genuine capability regression.

16.3 Benchmarks and Failure Modes

16.3.1 Capability Benchmarks

Public benchmarks give shared reference points. MMLU measures broad multiple-choice knowledge across academic subjects [54]. MATH tests competition-style mathematical problem solving [55]. HumanEval checks code-generation functional correctness [20]. DROP measures discrete reasoning over paragraphs [37]. MGSM probes multilingual chain-of-thought reasoning [152]. HELM argues for holistic evaluation across scenarios, metrics, and transparency artifacts [90]. GPQA targets graduate-level science questions designed to be difficult even with web access [143]. SWE-bench evaluates whether models can resolve real GitHub issues in real repositories [67]. Humanity's Last Exam and BrowseComp reflect a 2025 shift toward harder, more multimodal, and agentic evaluations at the frontier [128, 121].

These benchmarks are useful, but each measures a specific construct. MMLU does not measure conversational helpfulness. GPQA does not measure safe deployment. SWE-bench depends on repository setup, tests, and agent harness design. BrowseComp measures web-browsing research behavior under a particular protocol, not general truthfulness. Benchmark literacy means knowing what a benchmark excludes.

Emergence claims are benchmark claims. If a task looks random for small models and strong for large models, the evaluator should first ask whether the curve is being shaped by the metric, the prompt, the answer extractor, or the model series being compared [174]. For generative tasks, exact match may miss partial progress; for multiple-choice tasks, normalized likelihood and calibration can tell a different story from argmax accuracy. A useful evaluation card for an emergent ability plots every model point, includes chance and trivial baselines, reports confidence intervals, and preserves raw log probabilities or cross-entropy where possible.

16.3.2 pass@k and Selection Effects

Code and reasoning benchmarks often report pass@k: the probability that at least one of k generated samples passes. If a single sample has independent success probability p, then

$$\text{pass@}k = 1 - \text{pow}(1 - p, k).$$

Here $\text{pow}(1 - p, k)$ denotes the kth power of the single-sample failure probability. Independence rarely holds exactly, but the formula shows why pass@k is not the same as single-attempt reliability. A model with low pass@1 can look strong at pass@100 if sampling diversity is high and verification is cheap. For deployed agents, the relevant number may be pass@1 under a latency budget, pass@k with an automated verifier, or pass@k followed by human review.

16.3.3 Benchmark Contamination

LLMs are trained on large web corpora that may contain benchmark questions, answers, explanations, or near-duplicates. Contamination inflates scores and can make memorization look like reasoning. Surveys and detection methods document how difficult contamination analysis is for modern LLMs [153, 180]. Mitigations include private test sets, canary strings, deduplication against training corpora, transformed variants, time-split datasets, and post-release benchmark refreshes.

Contamination is not binary. A model may have seen the exact question, a paraphrase, a solution manual, a forum discussion, or many similar examples. Evaluation reports should describe the level of contamination control, not simply assert that a benchmark is clean.

16.3.4 Benchmark Saturation and Gaming

When a benchmark becomes influential, model developers tune for it. Scores rise, but the benchmark may stop distinguishing real deployment quality. Saturated benchmarks are still useful as regression tests: a model that fails easy public tasks has a problem. They are poor launch gates for frontier capability. Harder benchmarks such as GPQA, HLE, and BrowseComp are partly responses to saturation, but they too can become targets.

16.3.5 Long-Context, Agentic, and Generative Evaluation

Frontier systems increasingly fail outside the shape of short-answer benchmarks. Long-context models need tests for effective context use, not only maximum input length. ∞Bench, RULER, and LongBench v2 probe retrieval, aggregation, multi-hop reasoning, and realistic long-document understanding at context lengths where naive prompting can look impressive while actual evidence use remains brittle [199, 60, 10]. A long-context evaluation should report position sensitivity, distractor robustness, answer support, and cost per successful task.

Agentic evaluations add another layer. SWE-bench tests whether a model or agent can modify real repositories to resolve real issues [67]; BrowseComp-style evaluations test difficult web research under a constrained browsing protocol [121]. These are not just language tests. They evaluate model, tool runtime, state management, retry policy, sandboxing, and budget. A serious report should separate model capability from harness advantage: which tools were allowed, how many attempts were made, whether tests were visible, whether human hints were used, and how rollbacks were handled.

Generative multimodal systems need still different evidence. A text-to-image or text-to-video model should be evaluated for instruction following, visual fidelity, typography, object relations, temporal consistency, editing locality, diversity, safety filtering, and provenance. Unified understanding-generation systems need cross-task regression checks: improving image generation should not silently degrade OCR, document QA, speech latency, or refusal behavior. Janus-Pro, MMaDA, Sora-style video generation, and AR-Omni illustrate why one average score cannot represent a system that both understands and creates media [22, 186, 120, 25].

Video deserves its own caution. VBench++ dissects video generation into fine-grained dimensions such as temporal flicker, subject consistency, motion smoothness, spatial relationship, text-to-video, image-to-video, and trustworthiness [63]. Physics-oriented benchmarks such as T2VPhysBench ask whether generated videos respect first-principles constraints rather than merely looking plausible [51]. Video understanding benchmarks are also becoming more diagnostic: TOC-Bench targets temporal object consistency, requiring models to preserve object identity and state across occlusion, disappearance, reappearance, ordering, and interactions [19]. These evaluations should be read together. A system can be visually impressive, weak at physical consistency, and still poor at object-centric temporal reasoning.

Embodied and action systems add real-world accountability. A VLA benchmark should report task success, recovery after failure, unsafe action rate, intervention rate, latency, embodiment transfer, and whether test tasks share objects, scenes, operators, or teleoperation policies with training data. Simulation can accelerate evaluation, but it is not a substitute for real deployment evidence unless the sim-to-real gap is measured. For robotics, a low average failure rate may still be unacceptable if rare failures damage property or harm people.

16.4 Human Evaluation

16.4.1 Rubrics and Pairwise Preference

Human evaluation remains necessary when outputs are open-ended, values are contested, or task success is hard to automate. Pairwise preference is often more reliable than scalar scoring, but it needs a rubric. Annotators should know whether to prioritize factuality over style, safety over completeness, or concision over coverage. Ambiguous prompts should allow ties or “cannot judge” labels.

Report human evaluation with:

- number of prompts and comparisons;
- annotator qualifications and calibration process;
- rubric version;
- inter-annotator agreement;
- confidence intervals for win rates;
- examples of disagreements.

Without these details, a reported 55% win rate may be meaningful or noise.

16.4.2 LLM-as-Judge

LLM judges reduce cost and improve iteration speed. MT-Bench, Chatbot Arena, and related work show that strong models can approximate human preferences for many assistant comparisons [202, 26]. G-Eval and similar methods use structured prompts and rubrics to grade generated text [101]. These methods are useful for development, but they are not neutral measurement instruments.

Common judge biases include:

Position bias. Preferring the first or second answer because of ordering.
Verbosity bias. Preferring longer answers.
Self-preference. Preferring outputs from the judge's own model family.
Style bias. Rewarding polish over correctness.
Policy bias. Applying the judge model's safety policy rather than the target product policy.

Mitigations include answer-order randomization, length controls, judge ensembles, rubric-specific grading, calibration against human labels, and adversarial judge tests. For high-stakes launches, LLM judges should support human evaluation, not replace it.

16.5 Factuality, Hallucination, and Robustness

16.5.1 Truthfulness and Factual Precision

Truthfulness is not identical to confidence or fluency. TruthfulQA targets questions where imitation of common misconceptions leads to false answers [96]. FActScore decomposes long-form generations into atomic facts and measures support against a knowledge source [110]. HaluEval collects hallucination examples for detection and analysis [86]. RAG evaluation frameworks measure context relevance, answer faithfulness, and answer correctness for retrieval-augmented systems [38, 45].

For a generated answer A decomposed into atomic claims $c_1, \ldots, c_n$, a simple factual precision estimate is

$$S(A) = \{i \in \{1, \ldots, n\} : c_i \text{ is supported by evidence}\},$$
$$\text{FactPrecision}(A) = |S(A)|/n.$$

This metric emphasizes precision, not completeness. A model can be precise by saying very little. Deployment evaluation should pair factual precision with coverage, abstention, and usefulness.

16.5.2 Calibration and Abstention

Many applications need calibrated uncertainty. A calibrated model's stated confidence should match empirical correctness. If the model says it is 80% confident on many answers, about 80% should be correct. Calibration can be measured with reliability diagrams and expected calibration error, but LLM free-form confidence is unstable. Better signals often come from consistency across samples, verifier scores, retrieval support, or domain-specific checks.

Abstention is a behavior, not a failure, when the prompt lacks enough information. Evaluations should include unanswerable and underspecified prompts. Otherwise, models learn that every prompt must produce a confident answer.

16.5.3 Robustness

Robustness testing changes the input while preserving the intended task. Examples include paraphrases, typos, adversarial suffixes, distractor documents, misleading context, low-resource languages, unusual formatting, and distribution shifts over time. A robust model should not change a factual answer because the prompt uses a different dialect, and it should not follow malicious instructions in retrieved content.

For RAG and tool systems, robustness includes component failures. The retriever may return stale evidence, the tool may time out, a parser may fail, or an API may change. The evaluation harness should inject such failures and verify graceful degradation.

16.6 Safety Evaluation

16.6.1 Policy Taxonomy

Safety evaluation starts with a policy taxonomy. A typical taxonomy includes self-harm, violence, weapons, cyber abuse, fraud, privacy invasion, sexual content, hate and harassment, regulated advice, child safety, and illegal activity. Each category needs allowed, disallowed, and borderline examples. The taxonomy must also define how the

assistant should respond: refuse, redirect, provide high-level educational context, ask a clarifying question, or escalate.

Safety metrics should separate:

- unsafe compliance rate on disallowed requests;
- false refusal rate on benign requests;
- quality of safe redirection;
- jailbreak robustness;
- policy consistency across languages and modalities;
- tool-use safety, such as whether the model calls an external tool for a disallowed purpose.

A single "safety score" hides the tradeoff between over-refusal and under-refusal.

Safety taxonomies should also be language- and domain-aware. Baichuan 2's harmlessness evaluation describes a Chinese-English safety set spanning bias and discrimination, insults and profanity, illegal or unethical content, physical health, mental health, financial privacy, and sensitive topics [184]. The important pattern is not the exact taxonomy name, but the slice design: a bilingual or domain model needs safety probes written in the languages and risk contexts where it will be used, including categories such as health, finance, privacy, and politically sensitive content that are easy to under-sample in generic red-team sets.

16.6.2 Red Teaming

Red teaming searches for failures, not average-case performance. It should include manual experts, automated prompt mutation, multilingual attacks, role-play, prompt injection, tool-use attacks, long-context attacks, and multimodal attacks. Findings should be converted into regression tests, but the team should expect adversaries to adapt after deployment.

Frontier model system cards increasingly report safety evaluations, external red teaming, and risk-framework decisions. OpenAI's 2025 Preparedness Framework update and o3/o4-mini system card describe tracked risk categories and safeguards for reasoning models [123, 122]. Anthropic's Claude system cards and Responsible Scaling Policy describe capability thresholds and AI Safety Level safeguards [5, 6]. Such reports are useful evidence, but they should be read with attention to scope, omitted details, and residual uncertainty.

16.7 Governance

16.7.1 Documentation

Documentation turns evaluation into an auditable artifact. Model cards describe intended use, limitations, evaluation results, and ethical considerations [111]. Datasheets

and data statements document dataset motivation, collection, composition, preprocessing, recommended uses, and risks [46, 12]. For LLM systems, these ideas extend to system cards that cover model, tools, retrieval sources, filters, monitoring, and deployment constraints.

A useful LLM system card includes:

- model version, base model, and post-training summary;
- intended users and prohibited uses;
- data sources at a level compatible with privacy and security constraints;
- benchmark and in-domain evaluation results;
- safety policy version and refusal behavior;
- known limitations and failure modes;
- monitoring, logging, and incident response process.

16.7.2 Risk Management Frameworks

Governance frameworks provide structure for risk decisions. NIST AI RMF 1.0 organizes AI risk work around govern, map, measure, and manage functions [116]; the NIST Generative AI Profile adapts these ideas to generative AI risks such as hallucination, data privacy, misuse, and information integrity [117]. The European Union AI Act creates a risk-based legal framework with obligations for high-risk systems and general-purpose AI models [41]. These frameworks do not replace engineering judgment, but they force teams to document risk ownership, evaluation evidence, and post-deployment controls.

A concrete governance review should map every launch decision to an evaluation card. For example, a customer-support RAG assistant might require at least 95% retrieval recall on a private escalation set, factual precision above 98% on cited claims, unsafe-compliance rate below 0.5% on policy probes, false-refusal rate below 3% on benign sensitive prompts, and p95 latency below the product threshold. Each number needs an uncertainty interval. For a binary metric with k failures in n examples, a common normal-approximation variance term is

$$\mathrm{SE}^2 = \frac{\hat{p}(1-\hat{p})}{n}, \qquad \hat{p} = k/n,$$

where the standard error is the square root of this variance term. Low failure rates often require exact or bootstrap intervals rather than a normal approximation. A launch gate that ignores uncertainty can approve a model because the sample was too small to expose rare failures.

Operationally, governance also needs an owner and a rollback path. A release record should name the policy version, model version, retrieval index version, tool permissions, logging retention period, review queue, incident severity rubric, rollback trigger, and communication plan. Copyright and privacy review should happen before data enters training or retrieval: if a source cannot be redistributed, revoked, audited, or access-controlled, it should not be silently absorbed into a weight update.

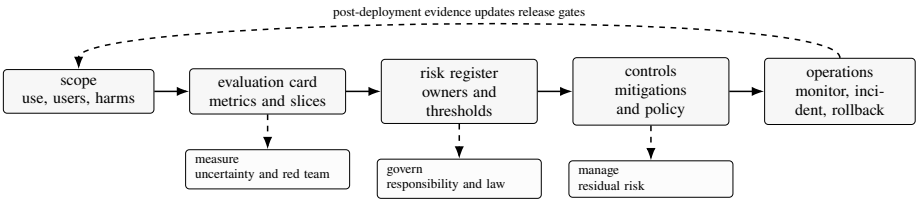

Fig. 16.2 Governance becomes actionable when framework language is mapped to release artifacts, owners, controls, monitoring, and rollback paths. The crosswalk synthesizes NIST AI RMF, the NIST Generative AI Profile, the EU AI Act, OpenAI governance frameworks, and Anthropic's RSP [116, 117, 41, 123, 124, 6].

Figure 16.2 shows the engineering crosswalk that turns broad governance frameworks into release artifacts.

16.7.3 Interpretability, Unlearning, and Watermarking

Governance needs technical levers, but none should be oversold. Mechanistic interpretability tries to identify features, circuits, or causal pathways inside a model. Anthropic's work on monosemantic features and circuit tracing shows that increasingly large models can sometimes be decomposed into interpretable computational structure, but these methods are still partial views rather than complete safety proofs [163, 4]. They are most useful when paired with interventions, counterexamples, and downstream behavior tests.

Unlearning addresses a different question: how can a model be made to stop producing or relying on targeted knowledge, private data, copyrighted material, or unsafe capabilities after training? Surveys of LLM unlearning emphasize that removal claims require careful evaluation of retained utility, extraction resistance, relearning risk, and collateral damage [134]. A model that stops answering one memorized prompt may still leak the same information through paraphrase, retrieval, or fine-tuning.

Watermarking and provenance tools aim to identify generated content or trace model outputs. Token-level watermarking schemes can bias sampling so generated text carries a detectable statistical signal [75]. For multimodal generation, provenance also includes metadata, content credentials, visible labels, model cards, and platform policy. NIST's synthetic-content transparency report treats authentication, provenance tracking, labeling, watermarking, detection, testing, and auditing as complementary technical approaches rather than a single silver bullet [18]. C2PA specifications define a technical standard for certifying the source and history of media content through content credentials and related provenance artifacts [28].

These tools are necessary but limited. Metadata can be stripped, visual watermarks can be cropped, statistical watermarks can be weakened by paraphrase or editing, and provenance chains can become ambiguous when generated assets are remixed. A governance policy should therefore layer signals: model-side watermarking, signed content

credentials, visible labeling where appropriate, platform detection, user reporting, and incident response. The release decision should state which signals are required for which modalities and what the system does when provenance is absent or conflicting.

16.7.4 Operational Monitoring

Pre-deployment evaluation is a snapshot. Deployed systems face changing users, prompts, tools, laws, and adversaries. Monitoring should track:

- traffic distribution and drift;
- refusal and escalation rates;
- user feedback and complaint categories;
- hallucination reports and citation failures;
- tool-call failures and suspicious tool use;
- latency, cost, and outage rates;
- safety incidents and near misses.

Monitoring must respect privacy. Logs should minimize sensitive data, enforce retention limits, and restrict access. For high-risk systems, human review workflows and incident response plans should exist before launch.

16.7.5 Deployment Limits

Some systems should not be deployed even if benchmark scores are high. Reasons include inadequate evaluation coverage, inability to monitor harms, high false-confidence risk, lack of domain oversight, legal restrictions, or severe misuse potential. A launch decision should weigh capability, uncertainty, reversibility, affected population, and available mitigations. “The model is better than the previous model” is not by itself a deployment argument.

16.8 Evaluation Checklist

Table 16.1 turns those release criteria into a compact review checklist.

16.9 Key Terms

Construct The underlying capability or behavior an evaluation intends to measure.

Evaluation harness The runnable code and configuration that turn samples, prompts, model calls, parsers, scorers, and aggregation into an auditable measurement.

Area	Question	Evidence
Task success	Does the system solve the user problem?	In-domain test set, human task review, tool success logs
Factuality	Are claims supported?	Atomic fact checks, retrieval faithfulness, abstention tests
Safety	Does it refuse the right requests?	Unsafe compliance and false refusal rates by category
Robustness	Does behavior survive perturbation?	Paraphrases, adversarial prompts, drift tests
Cost	Is the method deployable?	Latency, token use, hardware utilization, dollars per task
Governance	Can decisions be audited?	Model card, data documentation, policy version, incident process

Table 16.1 A compact checklist for LLM evaluation and deployment review.

Benchmark contamination Exposure of benchmark items or solutions during training or tuning.
pass@k Probability that at least one of k generated candidates passes an evaluation.
LLM-as-judge Use of a language model to score, rank, or critique outputs.
Hallucination Generated content that is unsupported or false in the relevant context.
Calibration Agreement between stated confidence and empirical correctness.
Red teaming Adversarial testing intended to discover failures.
Unlearning A post-training intervention intended to remove or suppress targeted knowledge or behavior while preserving unrelated capability.
Watermarking A method for embedding or detecting statistical or metadata signals that indicate generated content.
Content credentials Cryptographically signed provenance metadata describing the source, edits, or generative process associated with a media asset.
Model card Structured documentation of model use, evaluation, and limitations.
Risk management The process of identifying, measuring, mitigating, monitoring, and governing risks.

16.10 Exercises

1. Write an evaluation card for a small LLM application. Include construct, population, protocol, metric, uncertainty, and launch threshold.
2. Implement a tiny benchmark harness for ten multiple-choice questions. Save raw responses, parsed answers, scores, model version, and decoding settings as JSONL.
3. For a benchmark of your choice, list three ways contamination could occur and two mitigations that would still work after public release.
4. Run a pairwise human evaluation on twenty prompts with two model variants. Compute win rate and a bootstrap confidence interval.

5. Design an LLM-as-judge prompt for factual QA. Then list at least five judge biases and how you would test for them.
6. Design an evaluation card for a unified text-image model. Include separate metrics for understanding, generation, editing, safety, and provenance.
7. Design a video-generation evaluation that separates visual quality, prompt faithfulness, temporal consistency, physical consistency, and provenance.
8. Write a launch checklist for a VLA robot policy. Include simulation evidence, real-world trial evidence, unsafe-action handling, human intervention, and rollback.
9. Create a safety evaluation matrix with allowed, disallowed, and redirect examples for three policy categories.
10. Propose an unlearning evaluation for a small memorized corpus. Include extraction prompts, utility regression tests, and relearning checks.
11. Draft a one-page incident response plan for a deployed chatbot that produces unsafe or defamatory content.

Chapter 17
Research Frontiers and Practice Roadmap

Abstract This chapter connects the preceding technical material to a durable research and engineering roadmap. It emphasizes from-scratch implementation, data pipelines, resource accounting, emerging frontier model designs, reasoning reinforcement learning, adaptive test-time compute, multimodal and generative systems, agentic systems, and the operational evidence needed to turn new papers into dependable systems.

Chapter contract.

The reader should leave this chapter able to read frontier papers through the lifecycle built in earlier chapters, extract reproducible claims, map them to implementation and measurement work, and decide which ideas are durable enough to enter a curriculum or system roadmap.

17.1 Why Frontier Practice Changes the Curriculum

The preceding chapters explain the main lifecycle of a large language model: data, tokenization, architecture, pretraining, distributed systems, inference, adaptation, retrieval, alignment, reasoning, multimodality, generation, and evaluation. That lifecycle is stable enough to teach, but the frontier is not stable enough to freeze. The most important lesson from recent systems is that research progress often comes from changing the interface between chapters. A model report may describe an architectural change, but the observed gain may depend on the data mixture, optimizer, post-training recipe, evaluation harness, inference budget, and serving stack. A deployment result may look like an application result while actually being a systems result.

The unifying discipline is experimental falsifiability. A serious practitioner should be able to state the objective, implement a minimal version, account for compute and memory, measure data quality, run scaling or ablation experiments, report inference cost, and explain how the measurement could be wrong. Stanford's CS336, *Language*

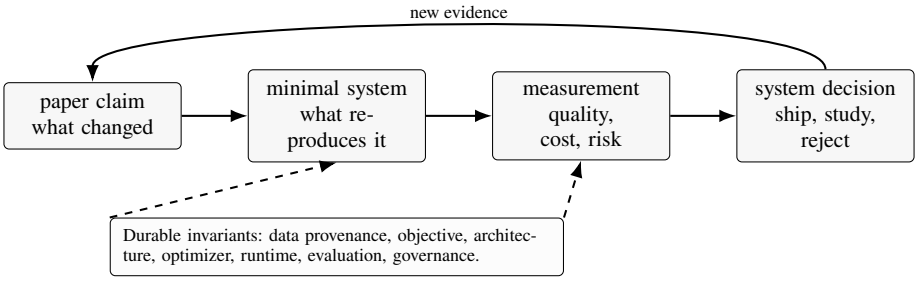

Fig. 17.1 Frontier practice is a loop from claim to minimal reproduction to measurement to system decision. Novel model names change quickly; the evidence loop should not.

Modeling from Scratch, makes this discipline explicit by organizing the course around tokenization, PyTorch and resource accounting, architecture and hyperparameters, attention alternatives and mixture-of-experts, GPUs and kernels, parallelism, scaling laws, inference, evaluation, data processing, and alignment or reasoning RL [157]. The same standard applies here: a chapter is complete only when the reader can connect a concept to an implementation, a measurement, and a failure mode.

Figure 17.1 summarizes the evidence loop for adding frontier claims to a durable curriculum.

17.2 Scope of the Roadmap

The roadmap covers the canonical decoder-only Transformer stack, LLaMA-class design, pretraining optimization, distributed training, inference serving, supervised instruction tuning, PEFT, domain adaptation, RAG, context engineering, typed memory, preference learning, reasoning, multimodality, generative foundation models, action interfaces, evaluation, safety, and governance. It also includes provenance and contamination concerns because modern foundation-model evidence is incomplete without them.

Splitting the field into separate treatments of LLMs, multimodal understanding, image/video generation, embodied agents, and governance would allow more local depth, but it would hide the central engineering fact: modern foundation-model work is a coupled lifecycle. The curriculum should therefore keep those topics connected. Multimodality changes context construction and serving; reasoning changes post-training and inference budgets; MoE changes training stability and schedulers; synthetic media changes evaluation and provenance; agents change safety and rollback.

First, implementation assignments should remain central. A modern reader benefits from writing a tokenizer, a small GPT, a masked SFT collator, a LoRA adapter, a retrieval evaluator, and a tiny verifier loop. These exercises should report not only task accuracy but also memory, FLOPs, tokens per second, and failure examples. The point is not to reproduce frontier scale. It is to make the invariants visible before scale hides them.

Second, systems analysis must keep expanding toward disaggregated inference and sparse models. Frontier serving increasingly adds mixture-of-experts routing, expert parallelism, prefill-decode disaggregation, cross-cluster routing, and heterogeneous resource allocation. The Frontier simulator paper is a useful example of how serving research is moving from dense co-located models to MoE and disaggregated designs [42]. A practical systems chapter should ask how model architecture changes the scheduler, not only how the scheduler serves a fixed model.

Third, the multimodal and generative chapter should be kept current. Qwen3-VL reports 256K-token long-context comprehension across text and interleaved multimodal inputs, stronger video and visual reasoning, and explicit temporal alignment mechanisms [137]. Qwen3-Omni pushes the interface further by unifying text, image, audio, and video understanding with speech generation and low-latency streaming considerations [182]. Janus-Pro, MMaDA, Dream 7B, Sora-style video generation, AR-Omni, RT-2, and OpenVLA show that the field is no longer only about attaching a vision encoder to a chatbot [22, 186, 190, 120, 25, 14, 72]. The conceptual lesson is that multimodality and generation change context construction, objective choice, latency, action safety, privacy, evaluation, and product contracts.

Fourth, the alignment and reasoning chapters should track the shift from preference imitation toward verifiable reinforcement learning and adaptive inference. DeepSeek-R1 showed that reasoning behavior can be incentivized through large-scale RL on verifiable tasks without relying entirely on human-written reasoning traces [33]. Qwen3 reports model families with reasoning and non-reasoning modes, multilingual breadth, and open release practices [183]. Kimi K2 frames open-weight progress around agentic coding, tool use, long context, sparse MoE, and post-training for agent behavior [73]. Surveys of RL for LLMs now treat PPO, DPO, GRPO, RLHF, RLAIF, and RLVR as one broader design space rather than isolated recipes [156].

17.3 From-Scratch Practice as a Research Method

From-scratch practice is not nostalgia. It is a defense against cargo-cult model building. A reader who has implemented byte-pair encoding understands why token fertility affects multilingual performance. A reader who has written attention kernels, even in simplified form, understands why shape conventions and memory layout matter. A reader who has profiled a training step understands why global batch size, activation checkpointing, optimizer state, and communication overlap cannot be chosen independently.

The minimal implementation path should look like this. Start with a tokenizer and record its behavior on English, code, math, and non-Latin scripts. Build a small decoder-only model and validate tensor shapes, mask semantics, residual connections, and initialization. Train on a small corpus and inspect loss by data source. Add sampling and observe how temperature, top-p, repetition penalties, and stop tokens change outputs. Add resource accounting: parameter count, activation memory, optimizer memory, FLOPs, arithmetic intensity, tokens per second, and wall-clock cost. Then add one

systems optimization at a time: fused attention, mixed precision, gradient checkpointing, data parallelism, or tensor parallelism.

Only after that path is clear should the reader study large model reports. The reports will become less mysterious. A claim about a larger vocabulary becomes a claim about token efficiency and embedding size. A claim about MoE becomes a claim about active parameters, router stability, expert load balance, communication, and serving cost. A claim about long context becomes a claim about positional generalization, KV memory, retrieval behavior, and latency. A claim about reasoning becomes a claim about reward design, exploration, verifier quality, inference tokens, and selection bias.

17.4 Frontier Architecture Questions

The book's architecture chapters emphasize decoder-only Transformers because they remain the reference design for most deployed language models. The frontier, however, is exploring at least five pressure points.

The first pressure point is sparse computation. MoE models can increase total capacity while keeping active computation per token lower than a dense model of equal total size. This changes every comparison table. A serious report must distinguish total parameters, active parameters, router strategy, number of selected experts, auxiliary losses, expert parallelism, and the hardware used for inference. Kimi K2-style systems make this point visible by combining very large total parameter counts with much smaller active parameter counts [73]. The relevant research question is not whether MoE is stronger in the abstract. It is when sparse routing produces better quality per dollar under realistic traffic.

The second pressure point is attention memory. Grouped-query attention and multi-query attention reduce KV-cache size, but new designs also explore latent attention, recurrent state, state-space models, and external memory. RetNet, Mamba, Mamba-2, and Titans show different ways to challenge standard attention scaling [160, 49, 32, 11]. The evaluation burden is high: efficient long-context mechanisms must demonstrate not only perplexity but also retrieval, aggregation, contradiction handling, instruction following, and post-training compatibility.

The third pressure point is native multimodality and action. A vision-language model that bolts a vision encoder onto a text model is easier to build, but a native multimodal system may handle interleaved video, audio, speech, tool outputs, and action tokens more coherently. The tradeoff is engineering complexity. Token budgets, streaming latency, embodiment, control frequency, modality-specific safety policies, and provenance become part of the architecture.

The fourth pressure point is generative objective choice. Autoregressive modeling gives a simple causal factorization and streaming interface. Diffusion and rectified-flow modeling provide strong tools for high-dimensional perceptual data and editing. Hybrid AR-diffusion systems try to combine semantic planning with high-fidelity synthesis [126, 39, 151]. A serious architecture chapter should therefore compare objectives, not only backbones.

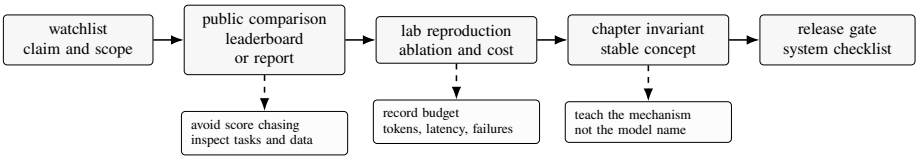

Fig. 17.2 Frontier material should move into the main text only after the claim has enough evidence to become a stable engineering invariant or release gate. The ladder reflects the book's use of CS336-style implementation evidence, public leaderboards, AILuminate-style safety benchmarks, and SWE-bench-style task harnesses [157, 64, 112, 162].

The fifth pressure point is trainability. Optimizers, initialization, normalization, clipping, and precision formats matter more as models become sparse, deeper, longer-context, multimodal, generative, and more heterogeneous. Architecture cannot be separated from the training run that makes it stable.

Figure 17.2 is the practical rule this chapter uses for deciding when a frontier result should reshape the book rather than remain on a reading list.

17.5 Reasoning and Adaptive Test-Time Compute

Reasoning is now a compute-allocation problem. A model can answer with a short response, produce a long trace, sample many candidates, call tools, run code, retrieve documents, search a tree, or ask a verifier to choose. These choices spend different kinds of budget. Test-time scaling surveys increasingly organize the field by what is scaled, how it is scaled, where in the pipeline scaling occurs, and how quality should be measured [197]. Budget-oriented surveys add another distinction: fixed controllability under a user-specified budget versus adaptive allocation based on difficulty, confidence, or intermediate evidence [3].

This book therefore treats a reasoning method as incomplete unless it states five things. First, what candidates are generated? Second, what scores or verifiers are trusted? Third, how much compute is spent and who controls that budget? Fourth, how does the method fail under adversarial or out-of-distribution prompts? Fifth, are gains due to a stronger policy, more search, better selection, benchmark contamination, or easier refusal behavior?

RLVR and GRPO-style training sharpen these questions. A reward that checks final answers can scale cheaply, but it may reward shortcuts. A process reward can guide intermediate steps, but labels are more expensive and can still be gamed. A group-relative baseline can simplify training by avoiding a learned critic, but it depends on group diversity. The same model can look better at pass@1 and worse at pass@k, or better under one token budget and worse under another. Evaluation must therefore report success, cost, latency, abstention, and failure modes together.

17.6 Data, Evaluation, and Governance as One Loop

The data chapter and the evaluation chapter should be read as one loop. Data filtering creates the distribution a model learns. Evaluation claims what the model can do. Governance decides which data and evaluations are allowed to influence deployment. When these are separated, teams can accidentally optimize toward contaminated benchmarks, unsafe synthetic data, or metrics that are detached from user risk.

A current data pipeline should record source licenses, collection dates, filtering rules, deduplication method, personally identifiable information policy, language distribution, document quality scores, synthetic-data generator versions, and benchmark overlap checks. A current evaluation pipeline should record prompt templates, decoding settings, tool permissions, retrieval indexes, judge models, human rubrics, confidence intervals, and failure slices. A current governance review should connect those artifacts to release decisions: what model is being launched, for whom, under what constraints, with what monitoring, and with what rollback path.

This is also where CS336-style implementation and frontier research meet. Data assignments that process Common Crawl, scaling assignments that fit loss curves, and alignment assignments that train reasoning models are not isolated course tasks. They are a compressed version of the lifecycle that real model teams must make auditable [157].

17.7 Frontier Evidence Matrix

New papers should enter the curriculum through evidence, not novelty pressure. A frontier claim is useful for this book when it changes a durable variable in the lifecycle and when that change can be reproduced, measured, and stress-tested. Table 17.1 summarizes the screening logic.

One way to keep the decision explicit is to score a claim as

$$S_{\text{frontier}} = \Delta Q - \lambda_C \Delta C - \lambda_R \Delta R - \lambda_U U + \lambda_E E, \tag{17.1}$$

where ΔQ is reproduced quality gain, ΔC is additional training or serving cost, ΔR is safety or governance risk, U is residual uncertainty, and E is independent evidence strength. The equation is not a universal metric. Its purpose is to prevent the curriculum from changing because a model name is new, a benchmark number is high, or a paper is fashionable before the underlying claim is stable.

17.8 A Reader's Roadmap After This Book

The reader should leave the book with four durable habits. First, reduce every model claim to a contract: objective, data, architecture, training recipe, inference policy, and evaluation protocol. Second, implement the smallest system that exposes the relevant

Table 17.1 How frontier directions should be incorporated into a durable curriculum.

Direction	Variables to track	Evidence needed before incorporation
Sparse MoE	Total parameters, active parameters, router, expert parallelism, serving latency	Better quality per dollar than dense baselines, with router stability and load-balance evidence
Long context and memory	Position mechanism, KV cost, retrieval, compression, conflicting evidence	More than accepting long inputs: evidence for locating, aggregating, and rejecting evidence
Reasoning RL and test-time compute	Reward, samples, verifier, token budget, pass@k/pass@1	Accuracy gains reported with latency, cost, trace policy, and verifier failure modes
Unified multimodal generation	Representation, objective, sampling steps, provenance controls	Joint evaluation of understanding, generation, temporal consistency, and safety boundaries
VLA and action models	Observation stream, action space, control frequency, interlocks, reset cost	Task success paired with unsafe-action rate and human-intervention policy

invariant. Third, compare models by task, cost, and risk rather than by name. Fourth, treat new papers as evidence to be tested, not as recipes to copy.

For further study, build projects in increasing order of coupling. Begin with a tokenizer and small GPT. Add a clean training loop and resource accounting. Add a data pipeline with deduplication and contamination checks. Add LoRA and SFT. Add a small RAG benchmark with retrieval recall and answer faithfulness. Add a preference model and DPO baseline. Add a verifiable math or code task and compare SFT, rejection sampling, and GRPO-style RL. Add an inference service with KV-cache measurement and latency percentiles. Add one multimodal component and evaluate perception separately from reasoning.

The frontier will change. This roadmap is meant to survive that change. If a future model family replaces attention, it will still need data, optimization, serving, adaptation, evaluation, and governance. If a future reasoning method replaces GRPO, it will still need reward design, budget accounting, and robustness tests. If a future multimodal system becomes native across all modalities, it will still need temporal grounding, privacy controls, action safety, provenance, and pipeline-level evaluation. The field rewards novelty, but engineering progress depends on preserving these invariants.

17.9 Key Terms

Adaptive test-time compute
: Inference that allocates different amounts of reasoning, search, retrieval, or verification to different inputs based on difficulty, uncertainty, or policy.

Disaggregated inference
: A serving architecture that separates stages such as prefill and decode, or attention and feed-forward computation, so they can scale on different resources.

From-scratch implementation	A learning and validation method in which key components are built directly to expose assumptions about data, tensors, optimization, and systems costs.
Generative foundation model	A large model trained to synthesize structured outputs such as text, image, audio, speech, video, code, or actions, often conditioned on multimodal context.
Native multimodality	A model or system design that handles modalities such as text, images, audio, video, and speech as first-class inputs or outputs rather than as late add-ons.
Resource accounting	Quantitative reporting of memory, FLOPs, bandwidth, tokens per second, latency, and cost for training or inference decisions.
VLA model	A vision-language-action model that uses multimodal observations and language instructions to produce actions for a physical or simulated environment.

17.10 Exercises

1. Choose one chapter of this book and design a CS336-style implementation assignment for it. Specify correctness tests, resource metrics, and failure analyses.
2. Compare a dense decoder model and an MoE model using total parameters, active parameters, routing overhead, KV-cache cost, and expected serving latency. Explain which number is most misleading if reported alone.
3. Design an adaptive test-time compute policy for math questions. State how the system decides between a short answer, long reasoning, sampling, tool use, and abstention.
4. Extend a RAG evaluation to include governance. Add fields for data license, access control, prompt-injection testing, citation faithfulness, and rollback criteria.
5. Redesign Chapter 15 as a standalone course module on unified understanding and generation. Specify which lectures cover autoregressive, diffusion, rectified-flow, and hybrid systems.
6. Extend the same module with a VLA lab. Specify the observation stream, action space, simulator or robot, success metric, safety interlock, and human intervention policy.
7. Pick one frontier paper from 2025 or later and identify which claim would fail if the evaluation used a different tokenizer, data mixture, inference budget, or benchmark split.

Appendix A
Appendix: Reproducibility and Notation Conventions

A.1 Notation

Vectors are written in bold lowercase, matrices in uppercase, and tensors with named shapes when ambiguity would otherwise obscure the implementation. Sequence length is denoted by T, hidden width by d_{model}, attention heads by h, and vocabulary size by V.

A.2 Experiment Records

Every executable experiment in the final manuscript should be accompanied by a compact run card. Table A.1 gives the minimum record. A chapter may add task-specific fields, but it should not omit the identity, data, execution, evaluation, and failure-analysis fields.

For short exercises, the run card can be a paragraph rather than a separate artifact. The important point is that another reader can identify the code version, data version, tokenizer and chat template, model configuration, random seed policy, optimizer, learning rate, batch size, precision, hardware, expected runtime, checkpoint and validation protocol, primary metric, observed failure modes, and the conclusion drawn from the run.

A.3 Source Provenance

The manuscript prose is original and cites public primary sources for research claims. It does not include copied third-party code, slide text, dataset rows, model outputs, or figures. Exercises are written as specifications that readers can implement themselves; when an exercise is inspired by a public project or paper, the relevant source is cited in the chapter bibliography.

Table A.1 Minimum run-card fields for reproducible manuscript experiments.

Field group	Required content
Run identity	Code version, configuration file, and command or notebook entry-point
Data contract	Dataset name, data version, license or provenance note, split, filtering, and contamination checks
Tokenization	Tokenizer version, special tokens, chat template, truncation, packing, and label-mask policy
Model contract	Model configuration, checkpoint source, adapter or quantization state, and context-length assumption
Optimization	Optimizer, learning rate schedule, batch size, precision, gradient accumulation, random seed policy, and stopping rule
Execution	Hardware, software stack, expected runtime, memory budget, checkpoint cadence, and resume semantics
Evaluation	Validation set, primary metric, secondary regression metrics, decoding or scoring settings, and uncertainty estimate
Failure analysis	Known failure modes, representative failed examples, safety or privacy caveats, and conclusion supported by the evidence

Glossary

Attention	A content-dependent mixing operation that forms query, key, and value representations and aggregates values according to query-key compatibility.
Benchmark contamination	Exposure of benchmark items, answers, or close variants during training, tuning, prompt development, or evaluation harness construction.
Context window	The maximum number of tokens a model can condition on during a forward pass.
Direct preference optimization	A preference-learning objective that fits a policy from chosen/rejected response pairs without training a separate online reward model.
Group relative policy optimization	A policy-optimization style that compares sampled responses within a group and uses their relative rewards as a baseline.
Grouped-query attention	Attention in which several query heads share each key-value head, reducing KV-cache memory relative to full multi-head attention.
Instruction tuning	Fine-tuning a pretrained model on examples that pair user-like instructions with target responses.
KV cache	Stored key and value tensors from previous decoding steps, reused to avoid recomputing attention over the prefix.
Large language model	A language model whose capability depends on the coupled scale of parameters, data, computation, post-training, and deployment system design.
Mixture of experts	A sparse architecture in which a router sends each token to a subset of expert networks, separating total parameters from active parameters.

Multi-head attention	Attention with several independent query, key, and value projection heads.
Multi-head latent attention	An attention variant that compresses part of the key-value state into latent representations for serving efficiency.
Multi-query attention	Attention in which all query heads share one key-value head, reducing decode-time KV-cache memory and bandwidth.
Parameter-efficient adaptation	Fine-tuning methods that update a small set of parameters or adapters while leaving most base-model weights frozen.
Preference model	A model or implicit objective that represents which of two candidate responses should be preferred under a target policy or annotation protocol.
Proximal policy optimization	A clipped policy-gradient method often used in RLHF to update a policy while limiting the size of each change.
Retrieval-augmented generation	A generation pattern in which external documents are retrieved and inserted into the model context before or during answer generation.
RLHF	Reinforcement learning from human feedback, usually involving preference data, reward modeling, and policy optimization under a reference-model constraint.
RLVR	Reinforcement learning with verifiable rewards, where correctness can be checked by an answer key, execution, proof checker, or task-specific verifier.
RoPE	Rotary position embeddings, which inject position by rotating query and key coordinates before attention scores are computed.
State-space model	A sequence model family that updates a compact state across positions, often used to reduce long-context cost relative to full attention.
Synthetic data	Training data generated or transformed by models or programs, often filtered by validators, reward models, or human audits.
Test-time compute	Additional computation spent at inference time through longer reasoning traces, multiple samples, verification, search, or refinement.
Tool use	A system pattern in which a model selects and conditions on external operations such as retrieval, code execution, calculators, browsers, or database queries.

References

1. Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebron, F., Sanghai, S.: GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv preprint arXiv:2305.13245 (2023). URL https://arxiv.org/abs/2305.13245
2. Alayrac, J.B., Donahue, J., Luc, P., Miech, A., Barr, I., Hasson, Y., Lenc, K., Mensch, A., Millican, K., Reynolds, M., Ring, R., Rutherford, E., Cabi, S., Han, T., Gong, Z., Samangooei, S., Monteiro, M., Menick, J., Borgeaud, S., Brock, A., Nematzadeh, A., Sharifzadeh, S., Binkowski, M., Barreira, R., Vinyals, O., Zisserman, A., Simonyan, K.: Flamingo: a Visual Language Model for Few-Shot Learning. arXiv preprint arXiv:2204.14198 (2022). URL https://arxiv.org/abs/2204.14198
3. Alomrani, M.A., Zhang, Y., Li, D., Sun, Q., Pal, S., Zhang, Z., Hu, Y., Ajwani, R.D., Valkanas, A., Karimi, R., et al.: Reasoning on a Budget: A Survey of Adaptive and Controllable Test-Time Compute in LLMs. arXiv preprint arXiv:2507.02076 (2025). URL https://arxiv.org/abs/2507.02076
4. Anthropic: Circuit tracing: Revealing computational graphs in language models. Transformer Circuits Thread (2025). URL https://transformer-circuits.pub/2025/attribution-graphs/methods.html
5. Anthropic: Claude 4 System Card. Tech. rep., Anthropic (2025). URL https://www.anthropic.com/system-cards
6. Anthropic: Responsible scaling policy version 3.3. Tech. rep., Anthropic (2026). URL https://www.anthropic.com/responsible-scaling-policy
7. Azar, M.G., Guo, Z.D., Piot, B., Munos, R., Rowland, M., Valko, M., Calandriello, D.: A general theoretical paradigm to understand learning from human preferences. International Conference on Artificial Intelligence and Statistics (2024). URL https://arxiv.org/abs/2310.12036
8. Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., et al.: Constitutional AI: Harmlessness from AI Feedback. arXiv preprint arXiv:2212.08073 (2022). URL https://arxiv.org/abs/2212.08073
9. Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., et al.: Training a helpful and harmless assistant with reinforcement learning from human feedback. arXiv preprint arXiv:2204.05862 (2022). URL https://arxiv.org/abs/2204.05862
10. Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., et al.: Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. arXiv preprint arXiv:2412.15204 (2024). URL https://arxiv.org/abs/2412.15204
11. Behrouz, A., Zhong, P., Mirrokni, V.: Titans: Learning to memorize at test time. arXiv preprint arXiv:2501.00663 (2025). URL https://arxiv.org/abs/2501.00663
12. Bender, E.M., Friedman, B.: Data statements for natural language processing: Toward mitigating system bias and enabling better science. Transactions of the Association for Computational Linguistics (2018). URL https://aclanthology.org/Q18-1041/

13. Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., van den Driessche, G., Lespiau, J.B., Damoc, B., Clark, A., et al.: Improving language models by retrieving from trillions of tokens. arXiv preprint arXiv:2112.04426 (2022). URL https://arxiv.org/abs/2112.04426
14. Brohan, A., Brown, N., Carbajal, J., Chebotar, Y., Chen, X., Choromanski, K., Ding, T., Driess, D., Dubey, A., Finn, C., et al.: Rt-2: Vision-language-action models transfer web knowledge to robotic control. arXiv preprint arXiv:2307.15818 (2023). URL https://arxiv.org/abs/2307.15818
15. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. arXiv preprint arXiv:2005.14165 (2020). URL https://arxiv.org/abs/2005.14165
16. Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J.D., Chen, D., Dao, T.: Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. arXiv preprint arXiv:2401.10774 (2024). URL https://arxiv.org/abs/2401.10774
17. Casper, S., Davies, X., Shi, C., Gilbert, T.K., Scheurer, J., Rando, J., Freedman, R., Korbak, T., Lindner, D., Freire, P., et al.: Open problems and fundamental limitations of reinforcement learning from human feedback. Transactions on Machine Learning Research (2023). URL https://arxiv.org/abs/2307.15217
18. Chandra, B., Dunietz, J., Roberts, K., Lee, Y., Fontana, P., Awad, G.: Reducing risks posed by synthetic content: An overview of technical approaches to digital content transparency. Tech. Rep. NIST AI 100-4, National Institute of Standards and Technology (2024). DOI 10.6028/NIST.AI.100-4
19. Chen, J., Meng, S., Chen, Y., Zhao, M., Gui, W., Guo, X.: Toc-bench: A temporal object consistency benchmark for video large language models. arXiv preprint arXiv:2605.09904 (2026). URL https://arxiv.org/abs/2605.09904
20. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021). URL https://arxiv.org/abs/2107.03374
21. Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174 (2016). URL https://arxiv.org/abs/1604.06174
22. Chen, X., Wu, Z., Liu, X., Pan, Z., Liu, W., Xie, Z., Yu, X., Ruan, C.: Janus-pro: Unified multimodal understanding and generation with data and model scaling. arXiv preprint arXiv:2501.17811 (2025). URL https://arxiv.org/abs/2501.17811
23. Chen, Y., Qian, S., Tang, H., Lai, X., Liu, Z., Han, S., Jia, J.: LongLoRA: Efficient Fine-tuning of Long-Context Large Language Models. International Conference on Learning Representations (2024). URL https://arxiv.org/abs/2309.12307
24. Chen, Z., Wang, S., Tan, Z., Fu, X., Lei, Z., Wang, P., Liu, H., Shen, C., Li, J.: A survey of scaling in large language model reasoning. arXiv preprint arXiv:2504.02181 (2025). URL https://arxiv.org/abs/2504.02181
25. Cheng, D., Yuan, R., Li, Y., You, R., Wang, W., Nie, L., Zhang, L., Li, W.: Ar-omni: A unified autoregressive model for any-to-any generation. arXiv preprint arXiv:2601.17761 (2026). URL https://arxiv.org/abs/2601.17761
26. Chiang, W.L., Zheng, L., Sheng, Y., Angelopoulos, A.N., Li, T., Li, D., Zhu, B., Zhang, H., Jordan, M.I., Gonzalez, J.E., Stoica, I.: Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference. arXiv preprint arXiv:2403.04132 (2024). URL https://arxiv.org/abs/2403.04132
27. Christiano, P.F., Leike, J., Brown, T.B., Martic, M., Legg, S., Amodei, D.: Deep reinforcement learning from human preferences. Advances in Neural Information Processing Systems (2017). URL https://arxiv.org/abs/1706.03741
28. Coalition for Content Provenance and Authenticity: C2PA Specifications. Technical specification (2026). URL https://spec.c2pa.org/specifications/specifications/2.0/index.html
29. Cui, Y., Yang, Z., Yao, X.: Efficient and Effective Text Encoding for Chinese LLaMA and Alpaca. arXiv preprint arXiv:2304.08177 (2023). URL https://arxiv.org/abs/2304.08177
30. Dao, T.: FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv preprint arXiv:2307.08691 (2023). URL https://arxiv.org/abs/2307.08691

31. Dao, T., Fu, D.Y., Ermon, S., Rudra, A., Re, C.: FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv preprint arXiv:2205.14135 (2022). URL https://arxiv.org/abs/2205.14135
32. Dao, T., Gu, A.: Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality. arXiv preprint arXiv:2405.21060 (2024). URL https://arxiv.org/abs/2405.21060
33. DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Wang, P., Xu, R., Zhu, Q., Zhang, R., Ma, S., et al.: DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv preprint arXiv:2501.12948 (2025). URL https://arxiv.org/abs/2501.12948
34. DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Ruan, C., et al.: DeepSeek-V3 Technical Report. arXiv preprint arXiv:2412.19437 (2024). URL https://arxiv.org/abs/2412.19437
35. Dettmers, T., Pagnoni, A., Holtzman, A., Zettlemoyer, L.: QLoRA: Efficient Finetuning of Quantized LLMs. arXiv preprint arXiv:2305.14314 (2023). URL https://arxiv.org/abs/2305.14314
36. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv preprint arXiv:1810.04805 (2018). URL https://arxiv.org/abs/1810.04805
37. Dua, D., Wang, Y., Dasigi, P., Stanovsky, G., Singh, S., Gardner, M.: DROP: A Reading Comprehension Benchmark Requiring Discrete Reasoning Over Paragraphs. Proceedings of NAACL-HLT (2019). URL https://arxiv.org/abs/1903.00161
38. Es, S., James, J., Espinosa-Anke, L., Schockaert, S.: RAGAS: Automated Evaluation of Retrieval Augmented Generation. arXiv preprint arXiv:2309.15217 (2023). URL https://arxiv.org/abs/2309.15217
39. Esser, P., Kulal, S., Blattmann, A., Entezari, R., Müller, J., Saini, H., Levi, Y., Lorenz, D., Sauer, A., Boesel, F., et al.: Scaling Rectified Flow Transformers for High-Resolution Image Synthesis. arXiv preprint arXiv:2403.03206 (2024). URL https://arxiv.org/abs/2403.03206
40. Ethayarajh, K., Xu, W., Muennighoff, N., Jurafsky, D., Kiela, D.: KTO: Model Alignment as Prospect Theoretic Optimization. arXiv preprint arXiv:2402.01306 (2024). URL https://arxiv.org/abs/2402.01306
41. European Union: Regulation (eu) 2024/1689 of the european parliament and of the council (artificial intelligence act) (2024). URL https://eur-lex.europa.eu/legal-content/en/TXT/?uri=CELEX:32024R1689
42. Feng, Y., Tan, X., Sew, K.H., Jiang, Y., Zhu, Y., Xu, H.: Frontier: Simulating the Next Generation of LLM Inference Systems. arXiv preprint arXiv:2508.03148 (2025). URL https://arxiv.org/abs/2508.03148
43. Frantar, E., Ashkboos, S., Hoefler, T., Alistarh, D.: GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv preprint arXiv:2210.17323 (2022). URL https://arxiv.org/abs/2210.17323
44. Fu, Y., Bailis, P., Stoica, I., Zhang, H.: Break the Sequential Dependency of LLM Inference Using Lookahead Decoding. arXiv preprint arXiv:2402.02057 (2024). URL https://arxiv.org/abs/2402.02057
45. Gan, A., Yu, H., Zhang, K., Liu, Q., Yan, W., Huang, Z., Tong, S., Hu, G.: Retrieval augmented generation evaluation in the era of large language models: A comprehensive survey. arXiv preprint arXiv:2504.14891 (2025). URL https://arxiv.org/abs/2504.14891
46. Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J.W., Wallach, H., Daumé III, H., Crawford, K.: Datasheets for datasets. Communications of the ACM (2021). URL https://cacm.acm.org/research/datasheets-for-datasets/
47. Gong, Y., Ran, D., Liu, J., Wang, C., Cong, T., Wang, A., Duan, S., Wang, X.: Figstep: Jailbreaking large vision-language models via typographic visual prompts. arXiv preprint arXiv:2311.05608 (2023). URL https://arxiv.org/abs/2311.05608
48. Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., et al.: The llama 3 herd of models. arXiv preprint arXiv:2407.21783 (2024). URL https://arxiv.org/abs/2407.21783
49. Gu, A., Dao, T.: Mamba: Linear-Time Sequence Modeling with Selective State Spaces. arXiv preprint arXiv:2312.00752 (2023). URL https://arxiv.org/abs/2312.00752

50. Guan, T., Liu, F., Wu, X., Xian, R., Li, Z., Liu, X., Wang, X., Chen, L., Huang, F., Yacoob, Y., Manocha, D., Zhou, T.: Hallusionbench: An advanced diagnostic suite for entangled language hallucination and visual illusion in large vision-language models. arXiv preprint arXiv:2310.14566 (2023). URL https://arxiv.org/abs/2310.14566
51. Guo, X., Huo, J., Shi, Z., Song, Z., Zhang, J., Zhao, J.: T2vphysbench: A first-principles benchmark for physical consistency in text-to-video generation. arXiv preprint arXiv:2505.00337 (2025). URL https://arxiv.org/abs/2505.00337
52. Gururangan, S., Marasovic, A., Swayamdipta, S., Lo, K., Beltagy, I., Downey, D., Smith, N.A.: Don't stop pretraining: Adapt language models to domains and tasks. Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (2020). URL https://arxiv.org/abs/2004.10964
53. Han, L., Mubarak, A., Baimagambetov, A., Polatidis, N., Baker, T.: Multimodal large language models: A survey. arXiv preprint arXiv:2506.10016 (2025). URL https://arxiv.org/abs/2506.10016
54. Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., Steinhardt, J.: Measuring massive multitask language understanding. arXiv preprint arXiv:2009.03300 (2020). URL https://arxiv.org/abs/2009.03300
55. Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., Steinhardt, J.: Measuring Mathematical Problem Solving With the MATH Dataset. arXiv preprint arXiv:2103.03874 (2021). URL https://arxiv.org/abs/2103.03874
56. Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L.A., Welbl, J., Clark, A., et al.: Training compute-optimal large language models. arXiv preprint arXiv:2203.15556 (2022). URL https://arxiv.org/abs/2203.15556
57. Holtzman, A., Buys, J., Du, L., Forbes, M., Choi, Y.: The curious case of neural text degeneration. International Conference on Learning Representations (2020). URL https://arxiv.org/abs/1904.09751
58. Hong, J., Lee, N., Thorne, J.: ORPO: Monolithic Preference Optimization without Reference Model. arXiv preprint arXiv:2403.07691 (2024). URL https://arxiv.org/abs/2403.07691
59. Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., de Laroussilhe, Q., Gesmundo, A., Attariyan, M., Gelly, S.: Parameter-efficient transfer learning for nlp. Proceedings of the 36th International Conference on Machine Learning (2019). URL https://arxiv.org/abs/1902.00751
60. Hsieh, C.P., Sun, S., Kriman, S., Acharya, S., Rekesh, D., Jia, F., Ginsburg, B.: Ruler: What's the real context size of your long-context language models? arXiv preprint arXiv:2404.06654 (2024). URL https://arxiv.org/abs/2404.06654
61. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: LoRA: Low-Rank Adaptation of Large Language Models. arXiv preprint arXiv:2106.09685 (2021). URL https://arxiv.org/abs/2106.09685
62. Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M.X., Chen, D., Lee, H., Ngiam, J., Le, Q.V., Wu, Y., Chen, Z.: Gpipe: Efficient training of giant neural networks using pipeline parallelism. Advances in Neural Information Processing Systems (2019). URL https://arxiv.org/abs/1811.06965
63. Huang, Z., Zhang, F., Xu, X., He, Y., Yu, J., Dong, Z., Ma, Q., Chanpaisit, N., Si, C., Jiang, Y., et al.: Vbench++: Comprehensive and versatile benchmark suite for video generative models. arXiv preprint arXiv:2411.13503 (2024). URL https://arxiv.org/abs/2411.13503
64. Hugging Face: Leaderboards and evaluations. Hugging Face documentation (2026). URL https://huggingface.co/docs/leaderboards/main/index
65. Hugging Face: LoRA. PEFT documentation (2026). URL https://huggingface.co/docs/peft/package_reference/lora
66. Hugging Face: Quantization. PEFT documentation (2026). URL https://huggingface.co/docs/peft/en/developer_guides/quantization
67. Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., Narasimhan, K.: Swe-bench: Can language models resolve real-world github issues? arXiv preprint arXiv:2310.06770 (2023). URL https://arxiv.org/abs/2310.06770

68. Kaplan, J., McCandlish, S., Henighan, T., Brown, T.B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., Amodei, D.: Scaling laws for neural language models. arXiv preprint arXiv:2001.08361 (2020). URL https://arxiv.org/abs/2001.08361
69. Karpathy, A.: nanoGPT. https://github.com/karpathy/nanoGPT (2022). GitHub repository
70. Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., Yih, W.t.: Dense passage retrieval for open-domain question answering. arXiv preprint arXiv:2004.04906 (2020). URL https://arxiv.org/abs/2004.04906
71. Keskar, N.S., McCann, B., Varshney, L.R., Xiong, C., Socher, R.: CTRL: A Conditional Transformer Language Model for Controllable Generation. arXiv preprint arXiv:1909.05858 (2019). URL https://arxiv.org/abs/1909.05858
72. Kim, M.J., Pertsch, K., Karamcheti, S., Xiao, T., Balakrishna, A., Nair, S., Rafailov, R., Foster, E., Lam, G., Sanketi, P., et al.: Openvla: An open-source vision-language-action model. arXiv preprint arXiv:2406.09246 (2024). URL https://arxiv.org/abs/2406.09246
73. Kimi Team, Bai, Y., Bao, Y., Charles, Y., Chen, C., Chen, G., Chen, H., Chen, J., Chen, N., Chen, R., et al.: Kimi k2: Open agentic intelligence. arXiv preprint arXiv:2507.20534 (2025). URL https://arxiv.org/abs/2507.20534
74. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014). URL https://arxiv.org/abs/1412.6980
75. Kirchenbauer, J., Geiping, J., Wen, Y., Katz, J., Miers, I., Goldstein, T.: A watermark for large language models. In: International Conference on Machine Learning (2023). URL https://arxiv.org/abs/2301.10226
76. Kudo, T., Richardson, J.: SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing. arXiv preprint arXiv:1808.06226 (2018). URL https://arxiv.org/abs/1808.06226
77. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C.H., Gonzalez, J.E., Zhang, H., Stoica, I.: Efficient memory management for large language model serving with pagedattention. arXiv preprint arXiv:2309.06180 (2023). URL https://arxiv.org/abs/2309.06180
78. Lambert, N., Pyatkin, V., Morrison, J., Miranda, L.J.V., Lin, B.Y., Chandu, K.R., Dziri, N., Kumar, S., Zick, T., Choi, Y., Smith, N.A., Hajishirzi, H.: Rewardbench: Evaluating reward models for language modeling. arXiv preprint arXiv:2403.13787 (2024). URL https://arxiv.org/abs/2403.13787
79. Lee, A.N., Hunter, C.J., Ruiz, N.: Platypus: Quick, Cheap, and Powerful Refinement of LLMs. arXiv preprint arXiv:2308.07317 (2023). URL https://arxiv.org/abs/2308.07317
80. Lee, H., Phatale, S.M.X., Lu, K., Mesnard, T., Bishop, C., Carbune, V., Rastogi, A.: RLAIF: Scaling Reinforcement Learning from Human Feedback with AI Feedback. arXiv preprint arXiv:2309.00267 (2023). URL https://arxiv.org/abs/2309.00267
81. Lee, K., Ippolito, D., Nystrom, A., Zhang, C., Eck, D., Callison-Burch, C., Carlini, N.: Deduplicating training data makes language models better. Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (2022). URL https://arxiv.org/abs/2107.06499
82. Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., Chen, Z.: Gshard: Scaling giant models with conditional computation and automatic sharding. arXiv preprint arXiv:2006.16668 (2020). URL https://arxiv.org/abs/2006.16668
83. Lester, B., Al-Rfou, R., Constant, N.: The power of scale for parameter-efficient prompt tuning. Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (2021). URL https://arxiv.org/abs/2104.08691
84. Leviathan, Y., Kalman, M., Matias, Y.: Fast Inference from Transformers via Speculative Decoding. Proceedings of the 40th International Conference on Machine Learning (2023). URL https://arxiv.org/abs/2211.17192
85. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Kuttler, H., Lewis, M., Yih, W.t., Rocktaschel, T., et al.: Retrieval-augmented generation for knowledge-intensive nlp tasks. arXiv preprint arXiv:2005.11401 (2020). URL https://arxiv.org/abs/2005.11401
86. Li, J., Cheng, X., Zhao, W.X., Nie, J.Y., Wen, J.R.: Halueval: A large-scale hallucination evaluation benchmark for large language models. Proceedings of the 2023 Conference on Empiri-

cal Methods in Natural Language Processing (2023). URL https://arxiv.org/abs/2305.11747
87. Li, J., Li, D., Savarese, S., Hoi, S.: BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models. arXiv preprint arXiv:2301.12597 (2023). URL https://arxiv.org/abs/2301.12597
88. Li, X.L., Liang, P.: Prefix-tuning: Optimizing continuous prompts for generation. Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics (2021). URL https://arxiv.org/abs/2101.00190
89. Li, Y., Du, Y., Zhou, K., Wang, J., Zhao, W.X., Wen, J.R.: Evaluating object hallucination in large vision-language models. arXiv preprint arXiv:2305.10355 (2023). URL https://arxiv.org/abs/2305.10355
90. Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., Yasunaga, M., Zhang, Y., Narayanan, D., Wu, Y., Kumar, A., Newman, B., Yuan, B., Yan, B., Zhang, C., Cosgrove, C., Manning, C.D., Re, C., Acosta-Navas, D., Hudson, D.A., Zelikman, E., Durmus, E., Ladhak, F., Rong, F., Ren, H., Yao, H., Wang, J., Santhanam, K., Orr, L., Zheng, L., Yuksekgonul, M., Suzgun, M., Kim, N., Guha, N., Chatterji, N., Khattab, O., Henderson, P., Huang, Q., Chi, R., Xie, S.M., Santurkar, S., Ganguli, S., Hashimoto, T., Icard, T., Zhang, T., Chaudhary, V., Wang, W., Li, X., Mai, Y., Zhang, Y., Koreeda, Y.: Holistic evaluation of language models. arXiv preprint arXiv:2211.09110 (2022). URL https://arxiv.org/abs/2211.09110
91. Liang, W., Liu, T.: Large Scale Transformer Model Training with Tensor Parallel. PyTorch Tutorials (2025). URL https://pytorch.org/tutorials/intermediate/TP_tutorial.html
92. Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., Cobbe, K.: Let's verify step by step. arXiv preprint arXiv:2305.20050 (2023). URL https://arxiv.org/abs/2305.20050
93. Lightning AI: Lit-LLaMA. https://github.com/Lightning-AI/lit-llama (2023). GitHub repository
94. Lin, B., Ye, Y., Zhu, B., Cui, J., Ning, M., Jin, P., Yuan, L.: Video-llava: Learning united visual representation by alignment before projection. arXiv preprint arXiv:2311.10122 (2023). URL https://arxiv.org/abs/2311.10122
95. Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.M., Wang, W.C., Xiao, G., Dang, X., Gan, C., Han, S.: AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. arXiv preprint arXiv:2306.00978 (2023). URL https://arxiv.org/abs/2306.00978
96. Lin, S., Hilton, J., Evans, O.: Truthfulqa: Measuring how models mimic human falsehoods. arXiv preprint arXiv:2109.07958 (2021). URL https://arxiv.org/abs/2109.07958
97. Liu, H., Li, C., Wu, Q., Lee, Y.J.: Visual instruction tuning. arXiv preprint arXiv:2304.08485 (2023). URL https://arxiv.org/abs/2304.08485
98. Liu, H., Tam, D., Muqeeth, M., Mohta, J., Huang, T., Bansal, M., Raffel, C.: Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. Advances in Neural Information Processing Systems (2022). URL https://arxiv.org/abs/2205.05638
99. Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., Liang, P.: Lost in the middle: How language models use long contexts. Transactions of the Association for Computational Linguistics (2023). URL https://arxiv.org/abs/2307.03172
100. Liu, X., Zhu, Y., Gu, J., Lan, Y., Yang, C., Qiao, Y.: Mm-safetybench: A benchmark for safety evaluation of multimodal large language models. arXiv preprint arXiv:2311.17600 (2023). URL https://arxiv.org/abs/2311.17600
101. Liu, Y., Iter, D., Xu, Y., Wang, S., Xu, R., Zhu, C.: G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment. arXiv preprint arXiv:2303.16634 (2023). URL https://arxiv.org/abs/2303.16634
102. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. International Conference on Learning Representations (2019). URL https://arxiv.org/abs/1711.05101
103. Lu, P., Bansal, H., Xia, T., Liu, J., Li, C., Hajishirzi, H., Cheng, H., Chang, K.W., Galley, M., Gao, J.: Mathvista: Evaluating mathematical reasoning of foundation models in visual contexts. arXiv preprint arXiv:2310.02255 (2023). URL https://arxiv.org/abs/2310.02255

104. Luo, R., Sun, L., Xia, Y., Qin, T., Zhang, S., Poon, H., Liu, T.Y.: BioGPT: Generative Pre-trained Transformer for Biomedical Text Generation and Mining. Briefings in Bioinformatics (2022). URL https://arxiv.org/abs/2210.10341
105. Lyu, Q., Havaldar, S., Stein, A., Zhang, L., Rao, D., Wong, E., Apidianaki, M., Callison-Burch, C.: Faithful chain-of-thought reasoning. arXiv preprint arXiv:2301.13379 (2023). URL https://arxiv.org/abs/2301.13379
106. Mei, L., Yao, J., Ge, Y., Wang, Y., Bi, B., Cai, Y., Liu, J., Li, M., Li, Z.Z., Zhang, D., et al.: A survey of context engineering for large language models. arXiv preprint arXiv:2507.13334 (2025). URL https://arxiv.org/abs/2507.13334
107. Meng, Y., Xia, M., Chen, D.: SimPO: Simple Preference Optimization with a Reference-Free Reward. arXiv preprint arXiv:2405.14734 (2024). URL https://arxiv.org/abs/2405.14734
108. Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., Wu, H.: Mixed precision training. arXiv preprint arXiv:1710.03740 (2017). URL https://arxiv.org/abs/1710.03740
109. Micikevicius, P., Stosic, D., Judd, P., Kamalu, J., Oberman, S., Shoeybi, M., Siu, M., Wu, H., Burgess, N., Ha, S., Grisenthwaite, R., Mellempudi, N., Cornea, M., Heinecke, A., Dubey, P.: FP8 Formats for Deep Learning. arXiv preprint arXiv:2209.05433 (2022). URL https://arxiv.org/abs/2209.05433
110. Min, S., Krishna, K., Lyu, X., Lewis, M., Yih, W.t., Koh, P.W., Iyyer, M., Zettlemoyer, L., Hajishirzi, H.: Factscore: Fine-grained atomic evaluation of factual precision in long form text generation. arXiv preprint arXiv:2305.14251 (2023). URL https://arxiv.org/abs/2305.14251
111. Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I.D., Gebru, T.: Model cards for model reporting. In: Proceedings of the Conference on Fairness, Accountability, and Transparency (2019). URL https://arxiv.org/abs/1810.03993
112. MLCommons: AILuminate. MLCommons documentation (2026). URL https://mlcommons.org/ailuminate/
113. Muennighoff, N., Yang, Z., Shi, W., Li, X.L., Fei-Fei, L., Hajishirzi, H., Zettlemoyer, L., Liang, P., Candès, E., Hashimoto, T.: s1: Simple test-time scaling. arXiv preprint arXiv:2501.19393 (2025). URL https://arxiv.org/abs/2501.19393
114. Mukherjee, S., Mitra, A., Jawahar, G., Agarwal, S., Palangi, H., Awadallah, A.: Orca: Progressive Learning from Complex Explanation Traces of GPT-4. arXiv preprint arXiv:2306.02707 (2023). URL https://arxiv.org/abs/2306.02707
115. Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., Zaharia, M.: Efficient large-scale language model training on gpu clusters using megatron-lm. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2021). URL https://arxiv.org/abs/2104.04473
116. National Institute of Standards and Technology: Artificial Intelligence Risk Management Framework (AI RMF 1.0). Tech. rep., National Institute of Standards and Technology (2023). URL https://nvlpubs.nist.gov/nistpubs/ai/NIST.AI.100-1.pdf
117. National Institute of Standards and Technology: Artificial intelligence risk management framework: Generative artificial intelligence profile. Tech. rep., National Institute of Standards and Technology (2024). URL https://nvlpubs.nist.gov/nistpubs/ai/NIST.AI.600-1.pdf
118. OpenAI: GPT-4o System Card. Tech. rep., OpenAI (2024). URL https://openai.com/index/gpt-4o-system-card/
119. OpenAI: simple-evals: Lightweight evaluation harnesses for language models. GitHub repository (2024). URL https://github.com/openai/simple-evals
120. OpenAI: Video generation models as world simulators. Technical report (2024). URL https://openai.com/research/video-generation-models-as-world-simulators
121. OpenAI: Browsecomp: A simple yet challenging benchmark for browsing agents. arXiv preprint arXiv:2504.12516 (2025). URL https://arxiv.org/abs/2504.12516
122. OpenAI: Openai o3 and o4-mini system card. Tech. rep., OpenAI (2025). URL https://openai.com/index/o3-o4-mini-system-card/

123. OpenAI: Our updated preparedness framework. Tech. rep., OpenAI (2025). URL https://openai.com/index/updating-our-preparedness-framework/
124. OpenAI: Openai's frontier governance framework. Tech. rep., OpenAI (2026). URL https://openai.com/index/openai-frontier-governance-framework/
125. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al.: Training language models to follow instructions with human feedback. arXiv preprint arXiv:2203.02155 (2022). URL https://arxiv.org/abs/2203.02155
126. Peebles, W., Xie, S.: Scalable Diffusion Models with Transformers. Proceedings of the IEEE/CVF International Conference on Computer Vision (2023). URL https://arxiv.org/abs/2212.09748
127. Peng, B., Quesnelle, J., Fan, H., Shippole, E.: Yarn: Efficient context window extension of large language models. arXiv preprint arXiv:2309.00071 (2023). URL https://arxiv.org/abs/2309.00071
128. Phan, L., Gatti, A., Han, Z., Li, N., Hu, J., Zhang, H., Zhang, C.B.C., Shaaban, M., Ling, J., Shi, S., Choi, M., et al.: Humanity's last exam. arXiv preprint arXiv:2501.14249 (2025). URL https://arxiv.org/abs/2501.14249
129. Press, O., Smith, N.A., Lewis, M.: Train short, test long: Attention with linear biases enables input length extrapolation. arXiv preprint arXiv:2108.12409 (2021). URL https://arxiv.org/abs/2108.12409
130. PyTorch Contributors: Asynchronous Saving with Distributed Checkpoint. PyTorch Tutorials (2026). URL https://docs.pytorch.org/tutorials/recipes/distributed_async_checkpoint_recipe.html
131. PyTorch Contributors: Distributed Checkpoint – torch.distributed.checkpoint. PyTorch 2.12 documentation (2026). URL https://docs.pytorch.org/docs/2.12/distributed.checkpoint.html
132. PyTorch Contributors: Pipeline Parallelism. PyTorch 2.12 documentation (2026). URL https://docs.pytorch.org/docs/2.12/distributed.pipelining.html
133. PyTorch Contributors: torch.distributed.fsdp.fully_shard. PyTorch 2.12 documentation (2026). URL https://docs.pytorch.org/docs/2.12/distributed.fsdp.fully_shard.html
134. Qiu, R., Tan, J., Pu, J., Wang, H., Gao, X.S., Sun, F.: A survey on unlearning in large language models. arXiv preprint arXiv:2510.25117 (2025). URL https://arxiv.org/abs/2510.25117
135. Qwen Team: Qwen2.5-Omni Technical Report. arXiv preprint arXiv:2503.20215 (2025). URL https://arxiv.org/abs/2503.20215
136. Qwen Team: Qwen2.5-VL Technical Report. arXiv preprint arXiv:2502.13923 (2025). URL https://arxiv.org/abs/2502.13923
137. Qwen Team: Qwen3-VL Technical Report. arXiv preprint arXiv:2511.21631 (2025). URL https://arxiv.org/abs/2511.21631
138. Radford, A., Kim, J.W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., Sutskever, I.: Learning transferable visual models from natural language supervision. Proceedings of the 38th International Conference on Machine Learning (2021). URL https://arxiv.org/abs/2103.00020
139. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners. OpenAI technical report (2019). URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
140. Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C.D., Finn, C.: Direct preference optimization: Your language model is secretly a reward model. arXiv preprint arXiv:2305.18290 (2023). URL https://arxiv.org/abs/2305.18290
141. Rajbhandari, S., Rasley, J., Ruwase, O., He, Y.: ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2020). URL https://arxiv.org/abs/1910.02054
142. Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., He, Y.: ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. Proceedings of the International Conference

for High Performance Computing, Networking, Storage and Analysis (2021). URL https://arxiv.org/abs/2104.07857
143. Rein, D., Hou, B.L., Stickland, A.C., Petty, J., Pang, R.Y., Dirani, J., Michael, J., Bowman, S.R.: GPQA: A Graduate-Level Google-Proof Q&A Benchmark. arXiv preprint arXiv:2311.12022 (2023). URL https://arxiv.org/abs/2311.12022
144. Ren, J., Rajbhandari, S., Aminabadi, R.Y., Ruwase, O., Yang, S., Zhang, M., Li, D., He, Y.: ZeRO-Offload: Democratizing Billion-Scale Model Training. USENIX Annual Technical Conference (2021). URL https://arxiv.org/abs/2101.06840
145. Rubenstein, P.K., Asawaroengchai, C., Nguyen, D.D., Bapna, A., Borsos, Z., de Chaumont Quitry, F., Chen, P., El Badawy, D., Han, W., Kharitonov, E., et al.: Audiopalm: A large language model that can speak and listen. arXiv preprint arXiv:2306.12925 (2023). URL https://arxiv.org/abs/2306.12925
146. Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., Scialom, T.: Toolformer: Language models can teach themselves to use tools. Advances in Neural Information Processing Systems (2023). URL https://arxiv.org/abs/2302.04761
147. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017). URL https://arxiv.org/abs/1707.06347
148. Sennrich, R., Haddow, B., Birch, A.: Neural machine translation of rare words with subword units. arXiv preprint arXiv:1508.07909 (2016). URL https://arxiv.org/abs/1508.07909
149. Shazeer, N.: Fast Transformer Decoding: One Write-Head is All You Need. arXiv preprint arXiv:1911.02150 (2019). URL https://arxiv.org/abs/1911.02150
150. Shazeer, N.: GLU Variants Improve Transformer. arXiv preprint arXiv:2002.05202 (2020). URL https://arxiv.org/abs/2002.05202
151. Shen, T., Wan, X., Chen, T., Zhang, R., Pan, J., Lu, D., Lei, F., Lu, Z., Yang, Y., Cheng, C., et al.: Mammothmoda2: A unified ar-diffusion framework for multimodal understanding and generation. arXiv preprint arXiv:2511.18262 (2025). URL https://arxiv.org/abs/2511.18262
152. Shi, F., Suzgun, M., Freitag, M., Wang, X., Srivats, S., Vosoughi, S., Chung, H.W., Tay, Y., Ruder, S., Zhou, D., et al.: Language models are multilingual chain-of-thought reasoners. arXiv preprint arXiv:2210.03057 (2022). URL https://arxiv.org/abs/2210.03057
153. Shi, W., Ajith, A., Xia, M., Huang, Y., Liu, D., Blevins, T., Chen, D., Zettlemoyer, L.: Detecting pretraining data from large language models. arXiv preprint arXiv:2310.16789 (2023). URL https://arxiv.org/abs/2310.16789
154. Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., Catanzaro, B.: Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053 (2019). URL https://arxiv.org/abs/1909.08053
155. Snell, C., Lee, J., Xu, K., Kumar, A.: Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters. arXiv preprint arXiv:2408.03314 (2024). URL https://arxiv.org/abs/2408.03314
156. Srivastava, S.S., Aggarwal, V.: A technical survey of reinforcement learning techniques for large language models. arXiv preprint arXiv:2507.04136 (2025). URL https://arxiv.org/abs/2507.04136
157. Stanford University: CS336: Language modeling from scratch. Course website (2026). URL https://cs336.stanford.edu/. Accessed 2026-05-20
158. Stiennon, N., Ouyang, L., Wu, J., Ziegler, D.M., Lowe, R., Voss, C., Radford, A., Amodei, D., Christiano, P.F.: Learning to summarize with human feedback. Advances in Neural Information Processing Systems (2020). URL https://arxiv.org/abs/2009.01325
159. Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., Liu, Y.: RoFormer: Enhanced Transformer with Rotary Position Embedding. arXiv preprint arXiv:2104.09864 (2021). URL https://arxiv.org/abs/2104.09864
160. Sun, Y., Dong, L., Huang, S., Ma, S., Xia, Y., Xue, J., Wang, J., Wei, F.: Retentive Network: A Successor to Transformer for Large Language Models. arXiv preprint arXiv:2307.08621 (2023). URL https://arxiv.org/abs/2307.08621
161. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, 2 edn. MIT Press (2018). URL http://incompleteideas.net/book/the-book-2nd.html

162. SWE-bench: SWE-bench Verified dataset card. Hugging Face dataset card (2026). URL https://huggingface.co/datasets/SWE-bench/SWE-bench_Verified
163. Templeton, A., Conerly, T., Marcus, J., Lindsey, J., Bricken, T., Chen, B., Pearce, A., Citro, C., Ameisen, E., Jones, A., et al.: Scaling Monosemanticity: Extracting Interpretable Features from Claude 3 Sonnet. Transformer Circuits Thread (2024). URL https://transformer-circuits.pub/2024/scaling-monosemanticity/
164. TorchTitan Contributors: TorchTitan: A PyTorch native platform for training generative AI models. GitHub repository (2026). URL https://github.com/pytorch/torchtitan
165. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Roziere, B., Goyal, N., Hambro, E., Azhar, F., et al.: LLaMA: Open and Efficient Foundation Language Models. arXiv preprint arXiv:2302.13971 (2023). URL https://arxiv.org/abs/2302.13971
166. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al.: Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023). URL https://arxiv.org/abs/2307.09288
167. Ud Din, M., Akram, W., Saoud, L.S., Rosell, J., Hussain, I.: Vision language action models in robotic manipulation: A systematic review. arXiv preprint arXiv:2507.10672 (2025). URL https://arxiv.org/abs/2507.10672
168. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. arXiv preprint arXiv:1706.03762 (2017). URL https://arxiv.org/abs/1706.03762
169. Wan, Z., Feng, X., Wen, M., Mcaleer, S.M., Wen, Y., Zhang, W., Wang, J.: Alphazero-like tree-search can guide large language model decoding and training. In: Proceedings of the 41st International Conference on Machine Learning, *Proceedings of Machine Learning Research*, vol. 235, pp. 49890–49920. PMLR (2024). URL https://proceedings.mlr.press/v235/wan24c.html
170. Wang, G., Qin, H., Jacobs, S.A., Holmes, C., Rajbhandari, S., Ruwase, O., Yan, F., Yang, L., He, Y.: ZeRO++: Extremely Efficient Collective Communication for Giant Model Training. arXiv preprint arXiv:2306.10209 (2023). URL https://arxiv.org/abs/2306.10209
171. Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E.H., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171 (2022). URL https://arxiv.org/abs/2203.11171
172. Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N.A., Khashabi, D., Hajishirzi, H.: Self-instruct: Aligning language models with self-generated instructions. arXiv preprint arXiv:2212.10560 (2022). URL https://arxiv.org/abs/2212.10560
173. Wei, J., Bosma, M., Zhao, V.Y., Guu, K., Yu, A.W., Lester, B., Du, N., Dai, A.M., Le, Q.V.: Finetuned language models are zero-shot learners. arXiv preprint arXiv:2109.01652 (2021). URL https://arxiv.org/abs/2109.01652
174. Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., et al.: Emergent abilities of large language models. Transactions on Machine Learning Research (2022). URL https://arxiv.org/abs/2206.07682
175. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. arXiv preprint arXiv:2201.11903 (2022). URL https://arxiv.org/abs/2201.11903
176. Wen, X., Liu, Z., Zheng, S., Ye, S., Wu, Z., Wang, Y., Xu, Z., Liang, X., Li, J., Miao, Z., et al.: Reinforcement Learning with Verifiable Rewards Implicitly Incentivizes Correct Reasoning in Base LLMs. arXiv preprint arXiv:2506.14245 (2025). URL https://arxiv.org/abs/2506.14245
177. Wu, X., Huang, C.C.: Introduction to context parallel. PyTorch Tutorials (2025). URL https://pytorch.org/tutorials/prototype/context_parallel.html
178. Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., Han, S.: Smoothquant: Accurate and efficient post-training quantization for large language models. Proceedings of the 40th International Conference on Machine Learning (2023). URL https://arxiv.org/abs/2211.10438
179. Xiao, G., Tian, Y., Chen, B., Han, S., Lewis, M.: Efficient streaming language models with attention sinks. International Conference on Learning Representations (2024). URL https://arxiv.org/abs/2309.17453

180. Xu, C., Guan, S., Greene, D., Kechadi, M.T.: Benchmark data contamination of large language models: A survey. arXiv preprint arXiv:2406.04244 (2024). URL https://arxiv.org/abs/2406.04244
181. Xu, C., Sun, Q., Zheng, K., Geng, X., Zhao, P., Feng, J., Tao, C., Jiang, D.: Wizardlm: Empowering large language models to follow complex instructions. arXiv preprint arXiv:2304.12244 (2023). URL https://arxiv.org/abs/2304.12244
182. Xu, J., Guo, Z., Hu, H., Chu, Y., Wang, X., He, J., Wang, Y., Shi, X., He, T., Zhu, X., et al.: Qwen3-Omni Technical Report. arXiv preprint arXiv:2509.17765 (2025). URL https://arxiv.org/abs/2509.17765
183. Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al.: Qwen3 Technical Report. arXiv preprint arXiv:2505.09388 (2025). URL https://arxiv.org/abs/2505.09388
184. Yang, A., Xiao, B., Wang, B., Zhang, B., Yin, C., Lv, C., Pan, D., Wang, D., Yan, D., Yang, F., et al.: Baichuan 2: Open large-scale language models. arXiv preprint arXiv:2309.10305 (2023). URL https://arxiv.org/abs/2309.10305
185. Yang, K., Ko, K., Ryu, M.: Tensor model parallelism tutorial. OSLO documentation (2025). URL https://oslo.eleuther.ai/TUTORIALS/tensor_model_parallelism.html
186. Yang, L., Tian, Y., Li, B., Zhang, X., Shen, K., Tong, Y., Wang, M.: Mmada: Multimodal large diffusion language models. arXiv preprint arXiv:2505.15809 (2025). URL https://arxiv.org/abs/2505.15809
187. Yang, Z., Guo, X., Zhang, T., Xu, H., Li, B.: Test-time Scaling of LLMs: A Survey from A Subproblem Structure Perspective. arXiv preprint arXiv:2511.14772 (2025). URL https://arxiv.org/abs/2511.14772
188. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T.L., Cao, Y., Narasimhan, K.: Tree of thoughts: Deliberate problem solving with large language models. arXiv preprint arXiv:2305.10601 (2023). URL https://arxiv.org/abs/2305.10601
189. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: React: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629 (2023). URL https://arxiv.org/abs/2210.03629
190. Ye, J., Xie, Z., Zheng, L., Gao, J., Wu, Z., Jiang, X., Li, Z., Kong, L.: Dream 7b: Diffusion large language models. arXiv preprint arXiv:2508.15487 (2025). URL https://arxiv.org/abs/2508.15487
191. Yoshida, D.: NF4 Isn't Information Theoretically Optimal (and That's Good). arXiv preprint arXiv:2306.06965 (2023). URL https://arxiv.org/abs/2306.06965
192. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., Radev, D.: Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (2018). URL https://arxiv.org/abs/1809.08887
193. Yue, X., Ni, Y., Zhang, K., Zheng, T., Liu, R., Zhang, G., Stevens, S., Jiang, D., Ren, W., Sun, Y., Wei, C., Yu, B., Yuan, R., Sun, R., Yin, M., Zheng, B., Yang, Z., Liu, Y., Huang, W., Sun, H., Su, Y., Chen, W.: Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi. arXiv preprint arXiv:2311.16502 (2023). URL https://arxiv.org/abs/2311.16502
194. Zelikman, E., Wu, Y., Mu, J., Goodman, N.: Star: Bootstrapping reasoning with reasoning. Advances in Neural Information Processing Systems (2022). URL https://arxiv.org/abs/2203.14465
195. Zhang, B., Sennrich, R.: Root mean square layer normalization. Advances in Neural Information Processing Systems (2019). URL https://arxiv.org/abs/1910.07467
196. Zhang, I., Liang, W.: Getting started with devicemesh. PyTorch Tutorials (2025). URL https://pytorch.org/tutorials/recipes/distributed_device_mesh.html
197. Zhang, Q., Lyu, F., Sun, Z., Wang, L., Zhang, W., Guo, Z., Wang, Y., King, I., Liu, X., Ma, C.: What, how, where, and how well? a survey on test-time scaling in large language models. arXiv preprint arXiv:2503.24235 (2025). URL https://arxiv.org/abs/2503.24235
198. Zhang, R., Han, J., Liu, C., Gao, P., Zhou, A., Hu, X., Yan, S., Pan, L., Li, H., Qiao, Y.: LLaMA-Adapter: Efficient Fine-tuning of Language Models with Zero-init Attention. arXiv preprint arXiv:2303.16199 (2023). URL https://arxiv.org/abs/2303.16199

199. Zhang, X., Chen, Y., Hu, S., Xu, Z., Chen, J., Hao, M.K., Han, X., Thai, Z.L., Wang, S., Liu, Z., Sun, M.: ∞bench: Extending long context evaluation beyond 100k tokens. arXiv preprint arXiv:2402.13718 (2024). URL https://arxiv.org/abs/2402.13718
200. Zhao, S., Zhang, X., Guo, J., Hu, J., Duan, L., Fu, M., Chng, Y.X., Wang, G.H., Chen, Q.G., Xu, Z., et al.: Unified multimodal understanding and generation models: Advances, challenges, and opportunities. arXiv preprint arXiv:2505.02567 (2025). URL https://arxiv.org/abs/2505.02567
201. Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., Li, S.: PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. Proceedings of the VLDB Endowment (2023). URL https://arxiv.org/abs/2304.11277
202. Zheng, L., Chiang, W.L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E.P., Zhang, H., Gonzalez, J.E., Stoica, I.: Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. arXiv preprint arXiv:2306.05685 (2023). URL https://arxiv.org/abs/2306.05685
203. Zheng, R., Dou, S., Gao, S., Hua, Y., Shen, W., Wang, B., Liu, Y., Jin, S., Liu, Q., Zhou, Y., et al.: Secrets of RLHF in Large Language Models Part I: PPO. arXiv preprint arXiv:2307.04964 (2023). URL https://arxiv.org/abs/2307.04964
204. Ziegler, D.M., Stiennon, N., Wu, J., Brown, T.B., Radford, A., Amodei, D., Christiano, P., Irving, G.: Fine-tuning language models from human preferences. arXiv preprint arXiv:1909.08593 (2019). URL https://arxiv.org/abs/1909.08593

Index